Содержание

Благодарности

Я хотел бы поблагодарить всех, кто выполнял технический обзор книги и внес в нее свой вклад: Феликс Линднер (Felix «FX» Lindner), Себастьян Крамер (Sebastian Krahmer), Дэн Розенберг (Dan Rosenberg), Фабиан Михайлович (Fabian Mihailowitsch), Стеффен Трешер (Steffen Tröscher), Андреас Курц (Andreas Kurtz), Марко Лоренц (Marco Lorenz), Макс Зиглер (Max Ziegler), Рене Шенфельдт (René Schönfeldt) и Силк Клейн (Silke Klein), а также Сондра Сильверхоук (Sondra Silverhawk), Элисон Ло (Alison Law) и всех остальных сотрудников издательства No Starch Press.

Введение

Добро пожаловать в книгу «Дневник охотника за ошибками». В этой книге описывается биография семи настоящих уязвимостей, обнаруженных мною за последние несколько лет. Каждой из них посвящена отдельная глава. В каждой главе я расскажу, как была обнаружена уязвимость, опишу шаги, позволяющие ее эксплуатировать, и как производитель исправил ее.

Цели этой книги

Главная цель этой книги – показать на практике, как вылавливать уязвимости. После ее прочтения вы будете лучше понимать приемы, используемые охотниками за ошибками при поиске уязвимостей безопасности, как создавать программный код, выявляющий и доказывающий существование уязвимостей, и как сообщить производителю об уязвимости.

Вторичная цель – рассказать историю каждой из семи описываемых здесь уязвимостей. Мне кажется, что они заслуживают вашего внимания.

Кому предназначена эта книга

Эта книга адресована исследователям и консультантам по проблемам безопасности программного обеспечения, программистам на C/C++, специалистам по выявлению уязвимостей и всем, кто желает погрузиться в захватывающий мир охоты за ошибками. Чтобы получить максимум от этой книги, читатель должен хорошо знать язык C и быть знаком с языком ассемблера для процессоров семейства x86.

Начинающих исследователей уязвимостей эта книга познакомит с разными аспектами их выявления, эксплуатации и составления отчетов для производителей программного обеспечения. Опытных охотников за ошибками эта книга познакомит с новыми взглядами на знакомые проблемы и иногда будет вызывать смех или появление снисходительной улыбки.

Отказ от ответственности

Цель этой книги – показать читателям, как выявлять уязвимости в программном обеспечении, защищаться от них и минимизировать их отрицательное влияние. Знание приемов выявления и эксплуатации уязвимостей совершенно необходимо, чтобы понимать основные проблемы и приемы защиты от них. С 2007 года, в Германии, где я живу, стало незаконным создавать или распространять «инструменты для взлома». К таким инструментам относятся простые сканеры портов, а также действующие эксплойты[1]. Поэтому, в соответствии с законом, в этой книге не будут представлены полные исходные тексты эксплойтов. Примеры, имеющиеся здесь, лишь демонстрируют шаги, позволяющие получить управление над потоком выполнения (над указателем инструкций или программным счетчиком) уязвимой программы.

Ресурсы

Все URL-адреса, упоминаемые в книге, а также примеры программного кода, информацию об ошибках и опечатках и другие сведения можно найти по адресу: http://www.trapkit.de/books/bhd/

1 Программы, эксплуатирующие уязвимости. – *Прим. перев.*

ГЛАВА 1

Выявление уязвимостей

Выявление уязвимостей – это процесс поиска ошибок в программном или аппаратном обеспечении. Однако в этой книге термин «выявление уязвимостей» используется исключительно для описания процесса поиска ошибок в программном обеспечении, вызывающих проблемы с безопасностью. Такие ошибки, также называемые уязвимостями, позволяют злоумышленнику проникать в удаленные системы, повышать уровень локальных привилегий, преодолевать границы привилегий или наносить иной ущерб системам.

Еще десять лет тому назад выявление уязвимостей в программном обеспечении воспринималось как хобби или средство привлечения внимания средств массовой информации. Когда стало понятно, что это занятие может приносить прибыль [1], к этому занятию стали относиться гораздо серьезнее.

Уязвимостям в программном обеспечении и программам, эксплуатирующим их (известным как эксплойты) уделяется большое внимание в прессе. Кроме того, существует огромное количество книг и ресурсов в Интернете, где описывается процесс эксплуатации уязвимостей и ведутся нескончаемые обсуждения о необходимости открытия информации об их выявлении. Но, не смотря на это, удивительно мало публикаций, описывающих сам процесс выявления. Термины, такие как «уязвимость в программном обеспечении» или «эксплойт» получили широкое распространение, однако многие – даже специалисты в области безопасности – не знают, как выявлять эти уязвимости.

Если спросить 10 разных охотников за ошибками, как они отыскивают уязвимости, вы наверняка получите 10 разных ответов. Это

одна из причин, почему нет и, скорее всего, никогда не будет готовых «рецептов» по выявлению уязвимостей. Вместо того, чтобы безуспешно пытаться составить универсальные инструкции, я расскажу в этой книге о подходах и приемах, использовавшихся для выявления ошибок в реальном программном обеспечении. Надеюсь, эта книга поможет читателю выработать свой стиль и самому выявить какие-нибудь необычные уязвимости.

1.1. Ради забавы и выгоды

Охотники за ошибками преследуют разные цели и имеют разную мотивацию. Одни стремятся повысить безопасность программного обеспечения, другие преследуют личную выгоду в виде известности, внимания средств массовой информации, денег или приглашения на работу. Компании могут стремиться выявлять уязвимости для использования этих фактов в маркетинговых целях. Разумеется, всегда найдется паршивая овца, ищущая новые пути к взлому систем или сетей. С другой стороны, некоторые занимаются этим исключительно ради забавы или, чтобы спасти мир.

1.2. Универсальные приемы

Не смотря на отсутствие официальной документации, описывающей стандартный процесс выявления уязвимостей, универсальные приемы все-таки существуют. Эти приемы можно разделить на две категории: статические и динамические. В статическом анализе (часто называется также статическим анализом программного кода) исследуются исходные тексты программ или результаты дизассемблирования двоичных файлов, но сами программы не запускаются. В динамическом анализе, напротив, активно используются приемы отладки и тестирования выполняющихся программ. Оба приема имеют свои достоинства и недостатки, и многие специалисты используют комбинации из статических и динамических приемов.

Мои личные предпочтения

Большей частью я предпочитаю статический анализ. Обычно я просматриваю исходные тексты или результаты дизассемблирования исследуемого программного обеспечения строка за строкой и пытаюсь

понять, как оно действует. Однако, читать весь программный код от начала до конца бессмысленно. При поиске ошибок я обычно стараюсь определить места, где в программу попадают пользовательские данные, введенные через интерфейс с внешним миром. В качестве примеров можно назвать данные, полученные из сети, из файла или из окружения времени выполнения.

Затем я изучаю пути, какими введенные данные следуют через программу, пока не найду потенциально уязвимый программный код, выполняющий операции над этими данными. Иногда такие точки входа удается выявить исключительно во время чтения исходных текстов (глава 2) или дизассемблированных листингов (глава 6). Иногда, чтобы отыскать код, обрабатывающий данные, в дополнение к статическому анализу приходится изучать результаты отладки исследуемого программного обеспечения (глава 5). Кроме того, при создании эксплойтов я обычно объединяю статический и динамический подходы.

После того, как ошибка будет найдена, необходимо доказать, что ею действительно можно воспользоваться, поэтому я пытаюсь написать для нее эксплойт. В процессе создания такого эксплойта, большую часть времени я провожу в отладчике.

Поиск потенциально уязвимого кода

Выше представлен лишь один из подходов, используемых при выявлении уязвимостей. Другая тактика поиска потенциально уязвимого кода заключается в изучении окрестностей вызовов «небезопасных» библиотечных функций языка C/C++, таких как `strcpy()` и `strcat()`, с целью выявить возможность переполнения буфера. В дизассемблерном листинге можно попробовать отыскать инструкции `movsx` и проверить возможность появления уязвимости переполнения знакового разряда. После обнаружения потенциально уязвимого кода, можно пойти по программному коду в обратном направлении и проследить, действительно ли выявленный фрагмент уязвим и достижим из точки входа в программу. Я редко пользуюсь этим приемом, но другие охотники за ошибками молятся на него.

Фаззинг

Существует совершенно иной подход к выявлению уязвимостей, известный как *фаззинг* (fuzzing). Фаззинг – это прием динамического

анализа, заключающийся в тестировании приложения посредством ввода недопустимых или ошибочных данных. Я не эксперт в фаззинге и в инструментах для его проведения, но я знаком со специалистами, создающими собственные инструменты для фаззинга и отыскавшими большое количество уязвимостей с их применением. Время от времени я использую этот прием, чтобы определить точки, где пользовательские данные попадают в программу и, иногда, для поиска уязвимостей (глава 8).

Возможно, кому-то будет интересно узнать, как можно использовать фаззинг для определения точек попадания пользовательских данных в программу. Представьте, что имеется сложное приложение в виде двоичного исполняемого файла, которое необходимо исследовать на наличие уязвимостей. Выявить точки ввода данных в таких сложных приложениях совсем непросто. Однако ввод недопустимых или ошибочных данных в сложных приложениях часто приводит к аварийному завершению. Это справедливо для программ, выполняющих анализ содержимого файлов, таких как офисные пакеты, аудио- и видеопроигрыватели или веб-браузеры. В большинстве случаев такие аварии не связаны с безопасностью (например, ошибка деления на ноль в браузере), но они часто позволяют определить начальную точку для исследования влияния пользовательских данных.

Дополнительная литература

Выше были описаны лишь несколько приемов и подходов к выявлению ошибок в программах. За дополнительной информацией о поиске уязвимостей в исходных текстах программ я рекомендую обратиться к книге Марка Дауда (Mark Dowd), Джона Макдональда (John McDonald) и Юстина Шу (Justin Schuh) «The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities» (Addison-Wesley, 2007). Желающим поближе познакомиться с приемом фаззинга можно порекомендовать книгу Майкла Саттона (Michael Sutton), Адама Грина (Adam Greene) и Педрама Амини (Pedram Amini) «Fuzzing: Brute Force Vulnerability Discovery» (Addison-Wesley, 2007)[1].

1 Педрам Амини, Майкл Саттон, Адам Грин «Fuzzing: исследование уязвимостей методом грубой силы», Символ-Плюс, 2009, ISBN: 978-5-93286-147-9. – *Прим. перев.*

1.3. Ошибки обращения с памятью

Уязвимости, описываемые в этой книге, объединяет одна общая черта: все они вызваны ошибками обращения с памятью. Такие ошибки возникают, когда процесс, поток выполнения или ядро пытается использовать:

- память, которой не владеет (например, разыменовать нулевой (NULL) указатель, как описывается в разделе A.2);
- больше памяти, чем было выделено (в результате возникает ошибка переполнения буфера, как описывается в разделе A.1);
- неинициализированную память (например, неинициализированные переменные) [2];
- механизм управления динамической памятью по ошибке (например, пытаться дважды освобождать один и тот же блок памяти) [3].

Ошибки обращения с памятью обычно происходят при неправильном использовании мощных возможностей языка C/C++, таких как явное управление памятью, или арифметические операции с указателями.

Ошибки обращения с памятью из подкатегории, которая называется порча содержимого памяти, возникают, когда процесс, поток выполнения или ядро изменяет содержимое памяти, которой не владеет, или когда изменения приводят к повреждению данных в памяти.

Тем, кто не знаком с такими ошибками, я предлагаю заглянуть в разделах A.1, A.2 и A.3 книги. Там описываются простейшие программные ошибки и уязвимости, обсуждаемые в этой книге.

Помимо ошибок обращения с памятью существуют десятки других классов уязвимостей. В их числе логические ошибки и веб-уязвимости, такие как межсайтовый скриптинг, подделка межсайтового запроса и инъекция SQL-кода. Однако эти классы уязвимостей не будут рассматриваться в данной книге. Все уязвимости, рассматриваемые здесь, являются результатом эксплуатации ошибок обращения с памятью.

1.4. Используемые инструменты

При выявлении уязвимостей или создании эксплойтов (для их тестирования), необходимо иметь возможность заглянуть внутрь выполняющегося приложения. Для этого я чаще всего использую отладчики и дизассемблеры.

Отладчики

Отладчик обычно предоставляет возможность присоединиться к пользовательскому процессу или к ядру, читать и изменять значения в регистрах и в памяти, и управлять потоком выполнения программы с помощью таких механизмов, как точки останова или пошаговый режим выполнения. Как правило, каждая операционная система распространяется вместе со своим собственным отладчиком, однако также имеются отладчики сторонних производителей. В табл. 1.1 перечислены различные операционные системы и отладчики, используемые в этой книге.

Таблица 1.1. Отладчики, используемые в этой книге

Операционная система	Отладчик	Отладка ядра
Microsoft Windows	WinDbg (официальный отладчик компании Microsoft) OllyDbg и его версия Immunity Debugger	да нет
Linux	GNU Debugger (gdb)	да
Solaris	Modular Debugger (mdb)	да
Mac OS X	GNU Debugger (gdb)	да
Apple iOS	GNU Debugger (gdb)	да

Эти отладчики будут использоваться для поиска, анализа и эксплуатации обнаруженных уязвимостей. Дополнительные сведения о некоторых командах отладчиков можно найти в разделах B.1, B.2 и B.4.

Дизассемблеры

Если необходимо исследовать приложение, исходные тексты которого недоступны, можно проанализировать двоичные файлы программы, изучив ее код на языке ассемблера. Отладчики обладают возможнос-

тью дизассемблировать исполняемый код процесса или ядра, однако пользоваться этой возможностью довольно сложно. Этот недостаток восполняет дизассемблер Interactive Disassembler Professional, более известный, как IDA Pro. [4] Дизассемблер IDA Pro поддерживает более 50 семейств процессоров, обеспечивает полную интерактивность, возможность расширения и графического представления структуры исполняемого кода. Дизассемблер IDA Pro совершенно необходим тем, кому требуется исследовать двоичные исполняемые файлы программ. Исчерпывающее описание IDA Pro можно найти в книге Криса Игла (Chris Eagle) «The IDA Pro Book, 2nd edition» (No Starch Press, 2011).

1.5. EIP = 41414141

Чтобы продемонстрировать влияние найденных мною ошибок на безопасность, я подробно буду рассказывать, какие шаги пришлось предпринять, чтобы получить контроль над потоком выполнения уязвимой программы, манипулируя указателем инструкций (Instruction Pointer, IP) процессора. Регистр указателя инструкций, или программный счетчик (Program Counter, PC), содержит смещение от начала текущего сегмента кода до инструкции, которая должна быть выполнена следующей. [5] Получая контроль над этим регистром, вы получаете безграничную возможность управления потоком выполнения уязвимого процесса. Для демонстрации захвата контроля над указателем инструкций, я буду изменять значение регистра, записывая в него, например, `0x41414141` (шестнадцатеричное представление строки ASCII-символов «AAAA»), `0x41424344` (шестнадцатеричное представление строки ASCII-символов «ABCD») или нечто подобное. Поэтому увидев в следующих главах выражение `EIP = 41414141`, знайте, что я получил контроль над уязвимым процессом.

Указатель инструкций/программный счетчик:

- EIP – 32-битный указатель инструкций (IA-32);
- RIP – 64-битный указатель инструкций (Intel 64);
- R15 или PC – архитектура ARM, используемая в устройствах iPhone компании Apple.*

* А также большинстве других смартфонов и мобильных устройств. – Прим. науч. ред.

После захвата контроля над указателем инструкций можно самыми разными способами направить его на подготовленный эксплойт.

За дополнительной информацией о процессе разработки эксплойтов можно обратиться к книге Джона Эриксона (Jon Erickson) «Hacking: The Art of Exploitation, 2nd edition» (No Starch Press, 2008)[2] или ввести строку «exploit writing» (написание эксплойтов) в Google и просмотреть огромное количество материалов, доступных в Интернете.

1.6. Заключительное примечание

В этой главе было рассмотрено множество основных понятий и у вас могло возникнуть множество вопросов. Не волнуйтесь, всему свое время. В следующих семи главах дневника дается более подробное описание тем, представленных здесь, и ответы на многие ваши вопросы. Можно также обратиться к приложениям, где приводится справочная информация по различным темам, обсуждаемым на протяжении всей книги.

Примечание. *Главы дневника следуют не в хронологическом порядке. Они расположены так, чтобы понятия в одной главе основывались на понятиях в другой.*

Примечания

1. Например: Педрам Амини (Pedram Amini), «Mostrame la guita! Adventures in Buying Vulnerabilities», 2009, http://docs.google.com/present/view?id=dcc6wpsd_20ghbpjxcr; Чарли Миллер (Charlie Miller), «The Legitimate Vulnerability Market: Inside the Secretive World of 0-day Exploit Sales», 2007, http://weis2007.econinfosec.org/papers/29.pdf; программа компании iDefense Labs выплаты вознаграждений за найденные уязвимости, https://labs.idefense.com/vcpportal/login.html; инициатива «Zero Day Initiative» компании TippingPoint, http://www.zerodayinitiative.com/.
2. См. презентацию Даниэля Ходсона (Daniel Hodson) «Uninitialized Variables: Finding, Exploiting, Automating» (Ruxcon, 2008),

2 Д. Эриксон, «Хакинг: искусство эксплойта, 2-е издание» (Символ-Плюс, 2009), ISBN: 978-5-93286-158-5. – Прим. перев.

http://felinemenace.org/~mercy/slides/RUXCON2008-UninitializedVariables.pdf.

3. См. статью «CWE-415: Double Free» на сайте Common Weakness Enumeration со списком типичных ошибок в разделе **CWE List ⇒ CWE - Individual Dictionary Definition (2.0)** по адресу: http://cwe.mitre.org/data/definitions/415.html.
4. http://www.hex-rays.com/idapro/.
5. См. руководство «Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture» по адресу: http://www.intel.com/products/processor/manuals/.

ГЛАВА 2

Назад в 90-е

Воскресенье, 12 октября, 2008

Дорогой дневник,

Сегодня я изучал исходные тексты популярного медиапроигрывателя VLC от проекта VideoLAN. Мне нравится проигрыватель VLC за его поддержку медиафайлов самых разных форматов и всех операционных систем, которыми я пользуюсь. Но поддержка большого количества форматов файлов имеет отрицательную сторону. Проигрыватель VLC вынужден проводить большой объем работ по парсингу файлов, что зачастую означает наличие большого числа ошибок, которые только и ждут, чтобы их обнаружили.

Примечание. *В книге «Parsing Techniques: A Practical Guide», Дика Грюна (Dick Grune) и Сэрил Дж. Х. Якобс (Ceriel J.H. Jacobs) [1] говорится: «Парсинг – это процесс структурирования линейного представления в соответствии с заданной грамматикой». Парсер – это программа, разбивающая последовательность байт на отдельные слова и предложения. В зависимости от формата представления данных, парсинг может оказаться весьма сложной процедурой, при реализации которой легко допустить ошибку.*

После ознакомления с внутренним устройством проигрывателя VLC, поиск первой уязвимости занял всего полдня. Это была классическая уязвимость переполнения буфера на стеке (раздел A.1). Данная уязвимость проявлялась при парсинге файлов в формате TiVo. Это проприетарный формат, используемый в устройствах видеозаписи, производимых компанией TiVo. Раньше я никогда не слышал об этом формате, но это не помешало мне эксплуатировать ошибку в его парсере.