**Exploiting / PWN. Автор: Раимов Хасан**

**Exploiting** или **Pwn** ("бинарная эксплуатация", "эксплойтинг", "пвн") — это категория задач, в которых, как правило, нужно искать и эксплуатировать уязвимости в скомпилированных приложениях. Чаще всего это уязвимости повреждения памяти (memory corruption).

Эта категория в каком-то смысле противопоставляется категории Web, в которой обычно стоит задача взломать приложение на языке высокого уровня (типа PHP), т.е. всё происходит на уровне интерпретации скрипта, а не на уровне физической памяти. Однако, порой в категорию Pwn включают и приложения на скриптовых языках. Это может быть, к примеру, обход песочницы (jail) на языке Python. Песочницей называют ограниченную среду, в которой разрешены только определённые команды. Целью решения таска обычно является обойти это ограничение и прочитать какую-то конфиденциальную информацию (флаг).

Надо отметить, что в последние годы тематика CTF-соревнований заметно укоренилась именно в области пвна. На топовых соревнованиях задачи зачастую очень сложные, и их решают известные профессионалы, исследователи, разработчики кибероружия. Например, одним из сильнейших пвнеров (и цтферов вообще) является <https://en.wikipedia.org/wiki/George_Hotz>, активно участвовавший в цтфах в 2013-2014 годах. Сильнейшей командой является команда PPP (Plaid Parliament of Pwning) родом из университета CMU, которая, как нетрудно догадаться, тоже специализируется в пвне.

Наиболее известным и распространённым случаем повреждения памяти является переполнение буфера. Если объяснить в двух словах, переполнение буфера возникает тогда, когда приложение принимает на вход строку и записывает её в некоторую выделенную область памяти (буфер), недостаточно проверяя её длину.

Таким образом, хакер может выйти за границы буфера и переписать какие-то иные участки памяти, в которых могут быть важные данные. Классический способ эксплуатации — это перетирание адреса возврата (return address). Это адрес памяти, на который программа "перепрыгнет" после завершения выполнения текущей функции.

Суть эксплойта в том, что хакер записывает в буфер любой код, который он хочет выполнить, далее выходит за границу буфера и меняет адрес возврата на адрес буфера. Таким образом, после выхода из текущей функции программа перепрыгнет на буфер, и выполнится код, который хакер туда записал.

Обычно в качестве условия задачи даётся бинарник (скомпилированный исполняемый файл программы) и адрес (хост и порт) сервера, на котором этот бинарник висит. Иногда даётся и исходный код приложения. Участнику нужно проанализировать приложение (дизассемблировать, декомпилировать или читать исходный код) и разработать эксплойт, т.е. программу, которая "заставляет" это уязвимое приложение выполнить произвольный код (вызвать шелл). Далее нужно при помощи этого эксплойта заставить сервер организаторов выдать флаг.

В простых случаях сам эксплойт может состоять всего из нескольких байт. Это может быть некоторое количество произвольных байтов для забивания буфера, а затем адрес какой-то функции, которая выдаёт флаг. Если этот адрес перезапишет адрес возврата, программа отдаст флаг.

Иногда даже для относительно непростых задач эксплойт пишется в одну строчку, но найти саму уязвимость и придумать цепочку вызовов для выполнения нужного действия не так легко.

Есть книга Д. Эриксона "Хакинг. Искусство эксплойта" в русском переводе. В ней очень доступно описано введение в эксплуатацию уязвимостей повреждения памяти. Также затронута сетевая безопасность и криптоанализ.

**Атака на переполнение буфера**

**Описание**

Ошибки переполнения буфера характеризуются перезаписью фрагментов памяти процесса, которые никогда не должны были быть изменены преднамеренно или непреднамеренно. Перезапись значений IP (указатель инструкции), BP (базовый указатель) и других регистров приводит к возникновению исключений, ошибок сегментации и других ошибок. Обычно эти ошибки неожиданно завершают выполнение приложения. Ошибки переполнения буфера возникают, когда мы работаем с буферами типа char.

Переполнение буфера может состоять из переполнения стека [Переполнение стека] или переполнения кучи [Переполнение кучи]. Мы не делаем различий между этими двумя в этой статье, чтобы избежать путаницы.

Нижеприведенные примеры написаны на языке C в системе GNU/Linux на архитектуре x86.

**Примеры**

**Пример 1**

#include <stdio.h>

int main(int argc, char \*\*argv)

{

char buf[8]; // buffer for eight characters

gets(buf); // read from stdio (sensitive function!)

printf("%s\n", buf); // print out data stored in buf

return 0; // 0 as return value

}

Это очень простое приложение считывает из стандартного ввода массив символов и копирует его в буфер типа char. Размер этого буфера составляет восемь символов. После этого содержимое буфера отображается и приложение закрывается.

Компиляция программы:

user@spin ~/inzynieria $ gcc bo-simple.c -o bo-simple

/tmp/ccECXQAX.o: In function `main':

bo-simple.c:(.text+0x17): warning: the `gets' function is dangerous and

should not be used.

На этом этапе даже компилятор предполагает, что функция gets() небезопасна.

Пример использования:

user@spin ~/inzynieria $ ./bo-simple // program start

1234 // we eneter "1234" string from the keyboard

1234 // program prints out the conent of the buffer

user@spin ~/inzynieria $ ./bo-simple // start

123456789012 // we eneter "123456789012"

123456789012 // content of the buffer "buf" ?!?!

Segmentation fault // information about memory segmenatation fault

Нам удается (не)удачно выполнить ошибочную операцию программой и спровоцировать ее аварийный выход.

Анализ проблемы:

Программа вызывает функцию, которая работает с буфером типа char и не делает проверок на переполнение размера, назначенного этому буферу. В результате можно преднамеренно или непреднамеренно сохранить в буфере больше данных, что вызовет ошибку. Возникает следующий вопрос: в буфере хранится всего восемь символов, так почему функция printf() вывела двенадцать? Ответ исходит из организации памяти процесса. Четыре символа, переполнившие буфер, также перезаписывают значение, хранящееся в одном из регистров, что было необходимо для корректного возврата функции. Непрерывность памяти привела к распечатке данных, хранящихся в этой области памяти.

**Пример 2**

#include <stdio.h>

#include <string.h>

void doit(void)

{

char buf[8];

gets(buf);

printf("%s\n", buf);

}

int main(void)

{

printf("So... The End...\n");

doit();

printf("or... maybe not?\n");

return 0;

}

Этот пример аналогичен первому. Кроме того, до и после функции doit() у нас есть два вызова функции printf().

Compilation:

user@dojo-labs ~/owasp/buffer\_overflow $ gcc example02.c -o example02

-ggdb

/tmp/cccbMjcN.o: In function `doit':

/home/user/owasp/buffer\_overflow/example02.c:8: warning: the `gets'

function is dangerous and should not be used.

Usage example:

user@dojo-labs ~/owasp/buffer\_overflow $ ./example02

So... The End...

TEST // user data on input

TEST // print out stored user data

or... maybe not?

Программа между двумя определенными вызовами printf() отображает содержимое буфера, который заполняется данными, введенными пользователем.

user@dojo-labs ~/owasp/buffer\_overflow $ ./example02

So... The End...

TEST123456789

TEST123456789

Segmentation fault

Поскольку размер буфера был определен (char buf[8]) и он был заполнен тринадцатью символами типа char, буфер переполнился.

Если наше бинарное приложение имеет формат ELF, то мы можем использовать программу objdump для его анализа и поиска необходимой информации для использования ошибки переполнения буфера.

Ниже приведен вывод, созданный objdump. Из этого вывода мы можем найти адреса, по которым вызывается printf() (0x80483d6 и 0x80483e7).

user@dojo-labs ~/owasp/buffer\_overflow $ objdump -d ./example02

080483be <main>:

80483be: 8d 4c 24 04 lea 0x4(%esp),%ecx

80483c2: 83 e4 f0 and $0xfffffff0,%esp

80483c5: ff 71 fc pushl 0xfffffffc(%ecx)

80483c8: 55 push %ebp

80483c9: 89 e5 mov %esp,%ebp

80483cb: 51 push %ecx

80483cc: 83 ec 04 sub $0x4,%esp

80483cf: c7 04 24 bc 84 04 08 movl $0x80484bc,(%esp)

80483d6: e8 f5 fe ff ff call 80482d0 <puts@plt>

80483db: e8 c0 ff ff ff call 80483a0 <doit>

80483e0: c7 04 24 cd 84 04 08 movl $0x80484cd,(%esp)

80483e7: e8 e4 fe ff ff call 80482d0 <puts@plt>

80483ec: b8 00 00 00 00 mov $0x0,%eax

80483f1: 83 c4 04 add $0x4,%esp

80483f4: 59 pop %ecx

80483f5: 5d pop %ebp

80483f6: 8d 61 fc lea 0xfffffffc(%ecx),%esp

80483f9: c3 ret

80483fa: 90 nop

80483fb: 90 nop

Если второй вызов printf() будет информировать администратора о выходе пользователя из системы (например, закрытая сессия), то мы можем попробовать пропустить этот шаг и закончить без вызова printf().

user@dojo-labs ~/owasp/buffer\_overflow $ perl -e 'print "A"x12

."\xf9\x83\x04\x08"' | ./example02

So... The End...

AAAAAAAAAAAAu\*.

Segmentation fault

Приложение закончило свое выполнение с ошибкой сегментации, но второй вызов printf() не состоялся.

Несколько слов пояснения:

perl -e 'print «A»x12 .\xf9\x83\x04\x08»' — выведет двенадцать символов «A», а затем четыре символа, которые на самом деле являются адресом инструкции, которую мы хотим выполнить. Почему двенадцать?

8 // size of buf (char buf[8])

+ 4 // four additional bytes for overwriting stack frame pointer

----

12

Анализ проблемы:

Проблема та же, что и в первом примере. Нет контроля над размером копируемого буфера в объявленный ранее. В этом примере мы перезаписываем регистр EIP адресом 0x080483f9, который на самом деле является вызовом ret на последней фазе выполнения программы.

Как по-другому использовать ошибки переполнения буфера?

Как правило, использование этих ошибок может привести к:

* приложение DoS
* изменение порядка выполнения функций
* выполнение кода (если мы сможем внедрить шеллкод, описанный в отдельном документе)

Как возникают ошибки переполнения буфера?

Такого рода ошибки очень легко сделать. В течение многих лет они были кошмаром программиста. Проблема заключается в собственных функциях C, которые не заботятся о соответствующих проверках длины буфера. Ниже приведен список таких функций и, если они существуют, их безопасные эквиваленты:

* gets() -\> fgets()- читать символы
* strcpy() -\> strncpy()- скопировать содержимое буфера
* strcat() -\> strncat()- объединение буферов
* sprintf() -\> snprintf()- заполнить буфер данными разных типов
* (f)scanf()- читать из STDIN
* getwd()- вернуть рабочий каталог
* realpath()- вернуть абсолютный (полный) путь

Используйте безопасные эквивалентные функции, которые проверяют длину буферов, когда это возможно. А именно:

1. gets() -\> fgets()
2. strcpy() -\> strncpy()
3. strcat() -\> strncat()
4. sprintf() -\> snprintf()

Те функции, которые не имеют безопасных эквивалентов, должны быть переписаны с реализованными безопасными проверками. Время, потраченное на это, принесет пользу в будущем. Помните, что вы должны сделать это только один раз.

Используйте компиляторы, которые умеют выявлять небезопасные функции, логические ошибки и проверять, не перезаписывается ли память там, где этого быть не должно.