ROP

Return Oriented Programming

Announcements

- DiceCTF next Friday (Feb 4)
  - Be there!
  - Let's be top 3
  - There will be pizza
- eCTF
  - Officially started
  - Check out #eCTF on Discord if signed up
- Research - talk to us after
- TracerFire March 6

Meeting Flag

sigpwny{nx_who?}

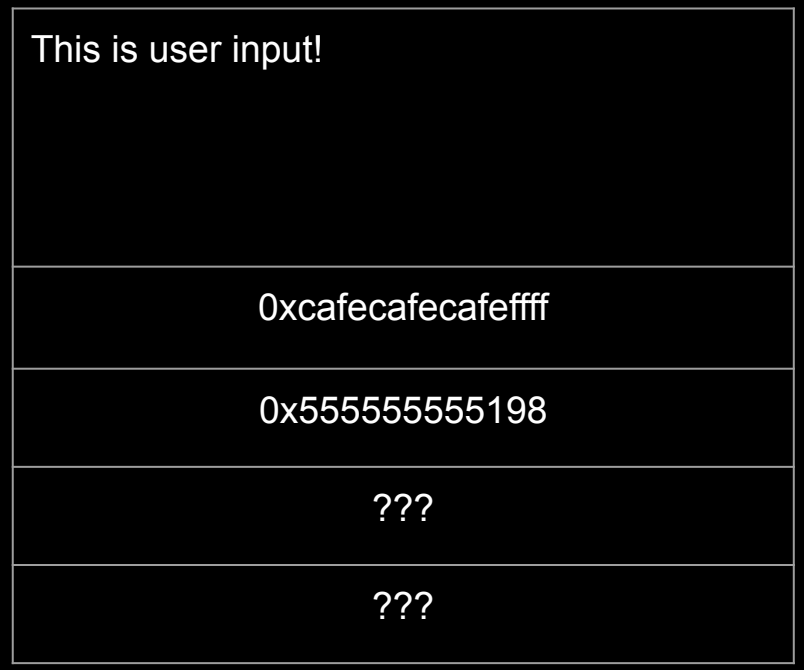
Overview

- Stack buffer overflow review
- W^X
- ROP high level
- ROP in practice

Vulnerable program

```
int main() {
    char buf[32];
    gets(buf);
}
```

Old way to solve



Lower address
(0xcafecafecafecafe)

Higher address
(0xcafecafecafecafe + 32+ 8 * 4)

```
int main() {
    char buf[32];
    gets(buf);
}
```

Old way to solve

buf (now full of instructions which open a shell)—>

Saved rbp —>

Return address —>

Other variables from previous function call

\xeb\x10\x48\x31\xc0\x5f\x48\x31 \xf6\x48\x31\xd2\x48\x83\xc0\x3b \x0f\x05\xe8\xeb\xff\xff\xff\x2f\x62 \x69\x6e\x2f\x2f\x73\x68\x90\x90
0x4141414141414141
0xcafecafecafecafe
???
???

Lower address (0xcafecafecafecafe)

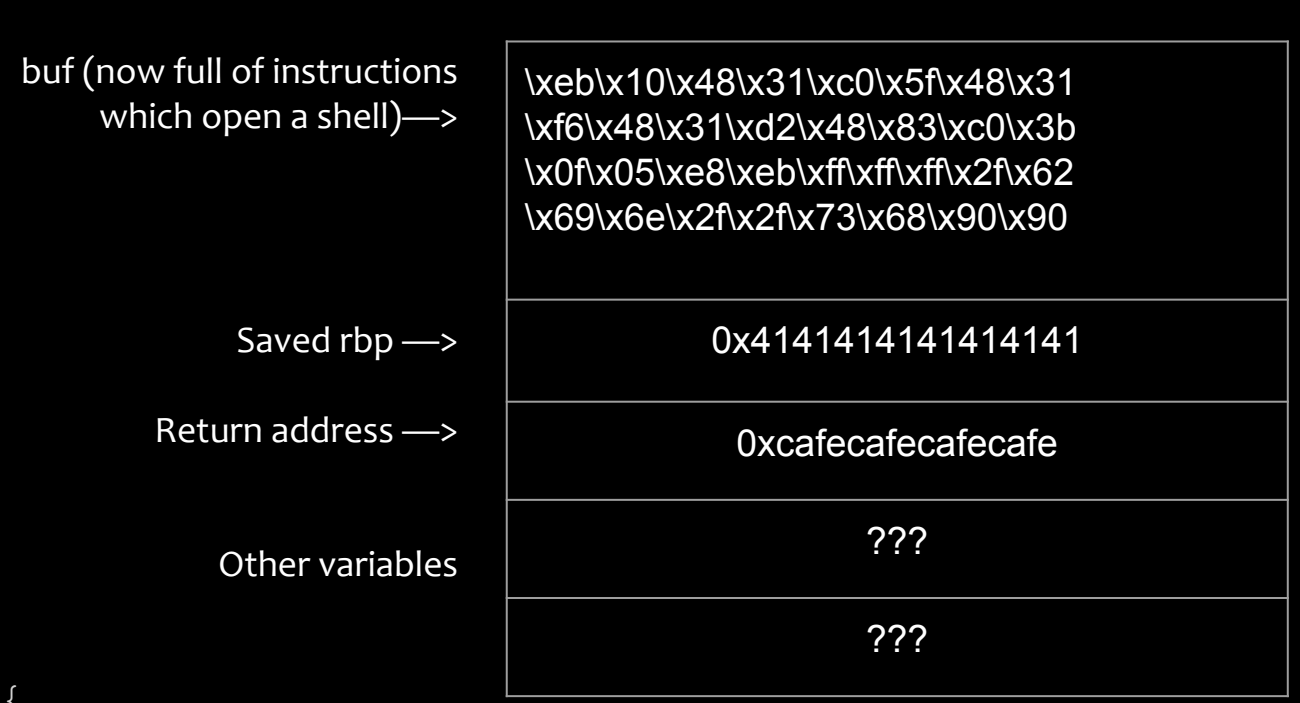
Higher address (0xcafecafecafecafe + 32+ 8 * 4)

```
int main() {
    char buf[32];
    gets(buf);
}
```

Introducing W^X

- Memory pages have permissions
  - R - read (Can you read the bytes of this memory)
  - W - write (Can you modify the bytes of this memory?)
  - X - execute (Can you jump to instructions in this memory?)
- Which permissions make most sense to apply to the stack?
- W^X philosophy
  - Write xor Execute
  - A memory page can be writable or executable, but should never be both at the same time

Old way to solve




```
int main() {
    char buf[32];
    gets(buf);
}
```

Now causes a SEGFAULT when jumping to oxcafecafecafecafe because stack memory is not executable!

ROP - Our Savior

- Code execution technique
  - Want to open a shell
- Bypasses NX (non executable) memory permissions
- Works by collecting “gadgets” and organizing them into a program

ROP - High Level

Gadget 1
A = A + 1

Gadget 2
A = 0

Gadget 3
B = A

Gadget 4
C = B

Using a sequence of gadgets, can we achieve:

B = 3

ROP - High Level

Gadget 1
A = A + 1

Gadget 2
A = 0

Gadget 3
B = A

Gadget 4
C = B

Using a sequence of gadgets, can we achieve:

B = 3

Gadget 2
Gadget 1
Gadget 1
Gadget 1
Gadget 3

ROP - High Level

Hint:
swap rax and rbx

Gadget 1
```
xchg rax, rbx
ret
```

Hint:
rbx = 0

Gadget 2
```
nop
xor rbx, rbx
ret
```

Hint:
rcx = 0
rax = rax + 1

Gadget 3
```
xor rcx, rcx
add rax, 1
ret
```

Hint:
rax = rax - rbx

Gadget 4
```
sub rax, rbx
nop
ret
```

Using a sequence of gadgets, can we achieve:

$$rbx = 3$$

(ignore the ret for now!)

ROP - High Level

Hint:
swap rax and rbx

Gadget 1
```
xchg rax, rbx
ret
```

Hint:
rbx = 0

Gadget 2
```
nop
xor rbx, rbx
ret
```

Hint:
rcx = 0
rax = rax + 1

Gadget 3
```
xor rcx, rcx
add rax, 1
ret
```

Hint:
rax = rax - rbx

Gadget 4
```
sub rax, rbx
nop
ret
```

Using a sequence of gadgets, can we achieve:

rbx = 3

(ignore the ret for now!)

Gadget 2 (set rbx to 0)
Gadget 1 (set rax = rbx)
Gadget 3 (rax = 1)
Gadget 3 (rax = 2)
Gadget 3 (rax = 3)
Gadget 1 (set rbx = rax)

ROP - High Level

1. Find gadgets in program
   a. Need gadgets that set registers
   b. Need gadget that invokes a syscall

2. Figure out how to order gadgets to set your registers to correct values for execve syscall

3. Execute your gadgets in order!

Where to find gadgets?

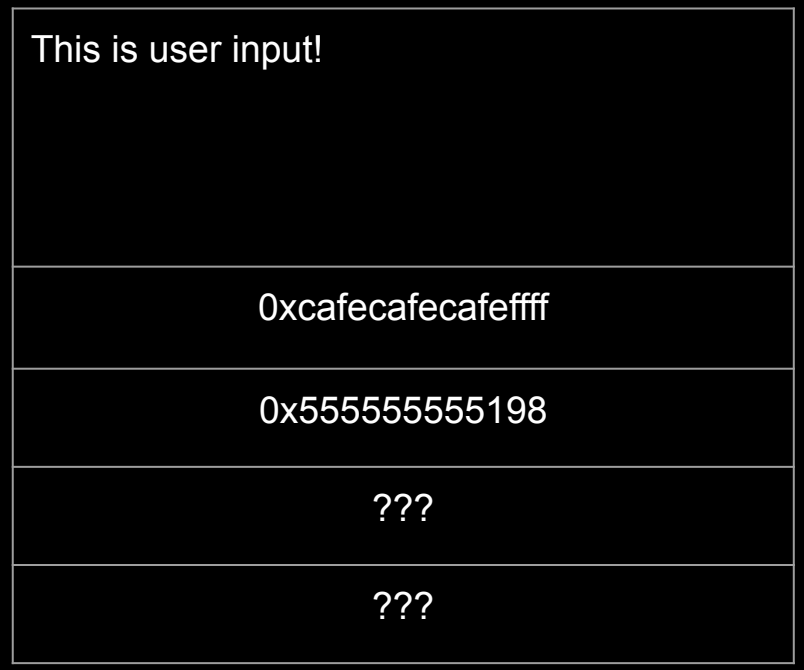
- Any instructions followed by a ‘ret’ is a gadget
  - May not be a useful gadget, though
  - objdump -d -M intel myprogram | grep ret -B 5

```
00000000000011e0 <__do_global_dtors_aux>:
    11e0:       f3 0f 1e fa             endbr64
    11e4:       80 3d 35 2e 00 00 00    cmp    BYTE PTR [rip+0x2e35],0x0
    11eb:       75 2b                   jne    1218 <__do_global_dtors_aux+0x38
    11ed:       55                      push   rbp
    11ee:       48 83 3d 02 2e 00 00    cmp    QWORD PTR [rip+0x2e02],0x0
    11f5:       00
    11f6:       48 89 e5                mov    rbp,rsp
    11f9:       74 0c                   je     1207 <__do_global_dtors_aux+0x27
    11fb:       48 8b 3d 06 2e 00 00    mov    rdi,QWORD PTR [rip+0x2e06]
    1202:       e8 a9 fe ff ff          call   10b0 <__cxa_finalize@plt>
    1207:       e8 64 ff ff ff          call   1170 <deregister_tm_clones>
    120c:       c6 05 0d 2e 00 00 01    mov    BYTE PTR [rip+0x2e0d],0x1
    1213:       5d                      pop    rbp
    1214:       c3                      ret
    1215:       0f 1f 00                nop    DWORD PTR [rax]
    1218:       c3                      ret
    1219:       0f 1f 80 00 00 00 00    nop    DWORD PTR [rax+0x0]
```

Vulnerable program - Second look

```
int main() {
    char buf[32];
    gets(buf);
}
```

Vulnerable program - Second look



Lower address
(0xcafecafecafecafe)

Higher address
(0xcafecafecafecafe + 32+ 8 * 4)

```
int main() {
    char buf[32];
    gets(buf);
}
```

ROP pwn


```
int main() {
    char buf[32];
    gets(buf);
}
```

ROP - Addresses

- Find the offset of gadget in the binary using objdump.
- Next, is PIE (position independent executable) enabled?
  - If yes: Need a leak to find base of binary
  - If no: Base of binary is always the same
    - fast way to find base is to just load with gdb, then `info file`
- Add base of binary to offset found with objdump
  - This is the memory address of the gadget which you should write on the stack

ROP - In practice

- You can find gadgets with objdump and hand craft gadget list

- … but most people just use ROPgadget
  - List gadgets
    - ./ROPgadget.py --binary myprogram
  - Automatically create a rop chain to pop shell
    - ./ROPgadget.py --ropchain --binary myprogram

ROP - Libc

- Small programs do not have enough gadgets to pop a shell

- No problem, just use libc
  - LOTS of gadgets
  - Basically all programs are linked with it, trick is finding correct version
    - Good chal authors give you the libc

1. Find gadgets in libc with ROPGadget/objdump
2. Leak libc address in running program
3. Calculate libc base from leak
4. Add gadget offset
5. Write addresses to stack

Next Meetings

Sunday Seminar:

- UIUCTF Planning

Next Thursday:

- Format string vulns