

(\$\$\$)

ixia



Hexcellents

Session 12

Return oriented programming

Security Summer School
31st of July 2014
ACS/Ixia/Hexcellents

Protection Mechanisms

- NX
- ASLR

Protection Mechanisms

- NX
- ASLR
- Bypass using `ret2libc`: call `mprotect` to 'reprotect' the stack with executable rights

Protection Mechanisms

- NX
- ASLR
- Bypass using `ret2libc`: call `mprotect` to 'reprotect' the stack with executable rights
- Issue 1: Some architectures (iOS) only run signed code. `mprotect` won't work

Protection Mechanisms

- NX
- ASLR
- Bypass using `ret2libc`: call `mprotect` to 'reprotect' the stack with executable rights
- Issue 1: Some architectures (iOS) only run signed code. `mprotect` won't work
- Issue 2: Using `ret2libc` we can call at most 2 functions. What if we need more?

Example

- We want to call `f1(0xAB, 0xCD)` and then `f2(0xEF, 0x42)`
- **RET+0x00:** addr of `f1`
RET+0x04: addr of `f2` (return address after `f1`)
RET+0x08: `0xAB` (param1 of `f1`)
RET+0x0c: `0xCD` (param2 of `f1`) **and** `0xEF` (param1 of `f2`)
RET+0x10: `0x42` (param2 of `f2`)

Example

- We want to call `f1(0xAB, 0xCD)` and then `f2(0xEF, 0x42)`
- **RET+0x00:** addr of `f1`
RET+0x04: addr of `f2` (return address after `f1`)
RET+0x08: `0xAB` (param1 of `f1`)
RET+0x0c: `0xCD` (param2 of `f1`) **and** `0xEF` (param1 of `f2`)
RET+0x10: `0x42` (param2 of `f2`)
- Value conflict

NOP Analogy

```
# objdump -d a -M intel | grep $'\t'ret
80482dd:          c3                ret
804837a:          c3                ret
80483b7:          c3                ret
8048437:          c3                ret
8048444:          c3                ret
80484a9:          c3                ret
80484ad:          c3                ret
80484c6:          c3                ret
```


NOP Analogy payload variant 1

RET + 0x00: 0x80482dd

RET + 0x04: 0x80482dd

RET + 0x08: 0x80482dd

RET + 0x0c: 0x80482dd

RET + 0x10: 0x80482dd

NOP Analogy payload variant 2

RET + 0x00: 0x80482dd

RET + 0x04: 0x804837a

RET + 0x08: 0x80483b7

RET + 0x0c: 0x8048437

RET + 0x10: 0x80484c6

Gadgets

- In ROP the basic building block is the `ret` instruction
- Any group of instructions that ends with `ret` is called a gadget

Gadgets

- In ROP the basic building block is the `ret` instruction
- Any group of instructions that ends with `ret` is called a gadget
- **0x8048443:** `pop ebp, ret`
- **0x80484a7:** `pop edi, pop ebp, ret`
- **0x8048441:** `mov ebp, esp, pop ebp, ret`
- **0x80482da:** `pop eax, pop ebx, leave, ret`
- **0x80484c3:** `pop ecx, pop ebx, leave, ret`

Gadgets

- In ROP the basic building block is the `ret` instruction
- Any group of instructions that ends with `ret` is called a gadget
- **0x8048443:** `pop ebp, ret`
- **0x80484a7:** `pop edi, pop ebp, ret`
- **0x8048441:** `mov ebp, esp, pop ebp, ret`
- **0x80482da:** `pop eax, pop ebx, leave, ret`
- **0x80484c3:** `pop ecx, pop ebx, leave, ret`

- What can we do with gadgets?

RETURN ORIENTED
PROGRAMMING

Pop to achieve a certain state

- Some architectures (x86_64, ARM, etc) use registers for function parameters
- Even simple exploits require ROP
- Let's see `system('/bin/sh')` 32 vs 64

Pop to achieve a certain state

- Some architectures (x86_64, ARM, etc) use registers for function parameters
- Even simple exploits require ROP
- Let's see `system('/bin/sh')` 32 vs 64
- Payload **x86**:
 - RET+0x00**: addr of `system`
 - RET+0x04**: JUNK
 - RET+0x08**: addr of `'/bin/sh'` (param1)

Pop to achieve a certain state

- Some architectures (x86_64, ARM, etc) use registers for function parameters
- Even simple exploits require ROP
- Let's see `system('/bin/sh')` 32 vs 64
- Payload **x86**:
 - RET+0x00**: addr of `system`
 - RET+0x04**: JUNK
 - RET+0x08**: addr of `'/bin/sh'` (param1)
- Payload **x86_64**:
 - RET+0x00**: addr of `'pop rdi; ret'` (rdi is param1)
 - RET+0x08**: addr of `'/bin/sh'`
 - RET+0x10**: addr of `system`

Pop to clear the stack

- Remember we wanted to call `f1(0xAB, 0xCD)` and then `f2(0xEF, 0x42)`
- **RET+0x00:** addr of `f1`
RET+0x04: addr of (`pop eax, pop ebx, ret`)
RET+0x08: `0xAB` (param1 of `f1`)
RET+0x0c: `0xCD` (param2 of `f1`)
RET+0x10: addr of `f2`
RET+0x14: JUNK
RET+0x18: `0xEF` (param1 of `f2`)
RET+0x1c: `0x42` (param2 of `f2`)

ROP is Turing complete

- It turns out you can actually execute anything given enough ROP gadgets
- <https://github.com/pakt/ropc>