Mitigation und ROP

Florian “Florob” Zeitz

Chaos Computer Club Cologne e.V.
https://koeln.ccc.de

2016-12-06

Outline

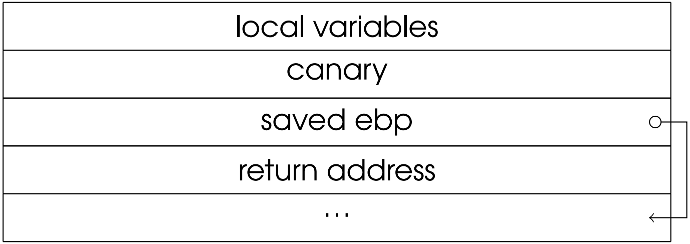
1. Mitigation
2. Return Oriented Programming

1. Mitigation
2. Return Oriented Programming

NX

- mark sections that only contain data as Non-eXecutable
- available per page on modern hardware (x86 with PAE, or AMD64)
- older x86 can use cs semgent as “line in the sand”
- extension: W^X
  - page can only be writable xor executable
  - SpiderMonkey: less than 3% performance penalty

Stack Smashing Protection



- check the current stack frame wasn't overwritten
- adds and checks an additional word
- `-fstack-protector{,-all,-strong,-explicit}`

Stack Smashing Protection: Implementation

- needs 2 symbols (usually from libc)
- `uintptr_t __stack_chk_guard;`
- `void __stack_chk_fail(void) __attribute__((noreturn));`

```
mov ($__stack_chk_guard), %eax
mov %eax, -0x1c(%ebp)
...
mov -0x1c(%ebp), %edi
xor ($__stack_chk_guard), %edi
jne bailout
...
bailout:
call __stack_chk_fail
```

Stack Smashing Protection: Canary Values

0xFF	'\n'	'\r'	0x00

^ time

random	*random*	*random*	*random*

uClibc canary

0x00	'\n'	0xff	0x00

0x00	*random*	*random*	*random*

glibc canary

Address Space Layout Randomization (ASLR)

- map memory segments to random addresses where possible
- mostly trivial for stack and heap (data)
- code cannot contain absolute addresses

Position Independent Code

- only use relative addressing
- usually explicitly or implicitly relative to `eip`
- relative jumps/calls
- norm for libraries, rare for executables
- get `eip` using a thunk

```
__x86.get_pc_thunk.bx:
mov (%esp), %ebx
ret
```

1 Mitigation

2 Return Oriented Programming

Motivation

- with NX we can’t execute shellcode in a data segment
- executable segments are not writable
- generally injecting our own code is hard
- What can we still achieve?

What can we still achieve?

- overwrite data, to influence control flow
- return into existing code

Returning into existing code

- can return to existing functions
- can return to the middle of them
- e. g. return to 0xb567 to change esp and return

```
f_call:
b550: sub    $0xc,%esp
b553: mov    0x14(%esp),%eax
b557: push   $0x0
b559: pushl  0x4(%eax)
b55c: pushl  (%eax)
b55e: pushl  0x1c(%esp)
b562: call   804ebd0 <luaD_call>
b567: add    $0x1c,%esp
b56a: ret
```

Splitting instructions

- can return to the middle of an instruction
- x86 instruction encoding is dense
- long instructions contain shorter ones
- immediates can contain instructions

```
b1f5: 83 c3 01        add  $0x1,%ebx
b1f8: e8 c3 ed ff ff  call printf

b1f7: 01 e8           add %ebp, %eax
b1f9: c3              ret
```

Return Oriented Programming

- few instructions followed by a `ret` are called *ROP gadget*
- chaining is possible by writing multiple gadget addresses on the stack

⇒ write shellcode by chaining ROP gadgets

Example: Write to arbitrary address

0xa804b2fc
0x0317112a
0xa804bd1a
0xbffff6d0
0xa805f104
...

```
a804b2fc:
pop %eax
ret
...
a804bd1a:
pop %ecx
ret
...
a805f104:
mov %eax, (%ecx)
ret
```

Finding gadgets

- gdb peda <https://github.com/longld/peda>

  dumprop <start> <end> [<keyword>] Dump gadgets in start:end
  dumprop <mapname> [<keyword>] Dump gadgets in mapname
  ropgadget [<mapname>] Print common gadgets (in mapname)
  ropsearch "<gadget>" <start> <end> Find gadget in start:end
  ropsearch "<gadget>" <mapname> Find gadget in mapname
  <mapname> can be e. g. binary or libc (any shared library)
- rp++ <https://github.com/0vercl0k/rp>
- ROPgadget <https://github.com/JonathanSalwan/ROPgadget>:
  ROPgadget --binary <bin>
- radare2: /R <gadget>

Aufgaben

- Challenge available as `/u23/rop/chksum`
- Try to get a shell!