Lec03: Stack Overflow

Taesoo Kim

Administrivia

- Survey: how many hours did you spend? (<3h, 6h, 10h, 15h, >20h)
- Please join Ed
- Optional recitations: Tue/Wed (@Coda)
- Lab03: stack overflow (finally!) challenges are out!
- Due : Sep 21 at midnight (2 weeks)

Survival Guide for CS6265

1. Work as a group/team (find the best ones around you!)
    - NOT each member tackles different problems
    - All members tackle the same problem (and discuss/help)
2. Ask questions wisely, concretely
    - Explain your assumption first (e.g., I expect A because …)
    - Explain your problem second (e.g., A is expected but B appears)
3. Take advantage of TAs standing next you to help!

Discussion 0

1. How different is the bomb binary this time?

Discussion 0

1. How different is the bomb binary this time?

- 64-bit (e.g., calling convention)

```
                  1     2    3      4    5    6
x86-64: func(%rdi, %rsi, %rdx, %r10, %r8, %r9, (%xmm0-7)) -> %rax
```

- int $80 vs. syscall? (`man syscall`)
- Stripped binary – no symbols
- Simple anti-debugging techniques

Discussion 1 (obfuscation)

- A linear disassembler does not work

```
(gdb) x/5i 0x4017b0
   0x4017b0:    jmp    0x4017b3 -------------?--+
   0x4017b2:    jmp    0x3f70a0                 V
   0x4017b7:    dec    DWORD PTR [rdi]
   0x4017b9:    (bad)
   0x4017ba:    test   BYTE PTR [rax],al
```

Discussion 1 (when running)


```
(gdb) x/5i 0x4017b3
   0x4017b3:    jmp    0x401710
   0x4017b8:    nop    DWORD PTR [rax+rax*1+0x0]
   0x4017c0:    push   rbp
   0x4017c1:    push   rbx
   0x4017c2:    mov    rbp,rdi
```

Discussion 2 (bomb203: signal)

1. What's going on the third phase?

- Messy control-flow with signal handling
- SIGTRAP by `int3`
- `handle SIGTRAP nostop` in gdb

Discussion 3 (bomb204: minfuck)

1. What's going on the last phase?

- simplified `brainfuck` interpreter
- Nothing special!
- What about dynamically testing in gdb?

How about poly shellcode?

1. What's your general idea?

Discrepancy b/w 32 vs 64

2.2.1.2 More on REX Prefix Fields

REX prefixes are a set of 16 opcodes that span one row of the opcode map and occupy entries 40H to 4FH. These opcodes represent valid instructions (INC or DEC) in IA-32 operating modes and in compatibility mode. In 64-bit mode, the same opcodes represent the instruction prefix REX and are not treated as individual instructions.

The single-byte-opcode forms of the INC/DEC instructions are not available in 64-bit mode. INC/DEC functionality is still available using ModR/M forms of the same instructions (opcodes FF/0 and FF/1).

See Table 2-4 for a summary of the REX prefix format. Figure 2-4 though Figure 2-7 show examples of REX prefix fields in use. Some combinations of REX prefix fields are invalid. In such cases, the prefix is ignored. Some additional information follows:

Dispatching routine


```
          +-------------+
          |             v
[dispatcher][x86      ][x86_64   ]

e.g., 0x40 0x90
 - x86       inc eax
 - x86_64    REX + nop

x86   : [ * ][goto x86 shellcode]
x86-64: [nop][ * ][goto x86_64 shellcode]
arm   : [nop][nop][ * ][goto arm shellcode]
MIPS  : [nop][nop][nop][ * ][goto MIPS shellcode]
```

Ref. http://ref.x86asm.net/geek.html

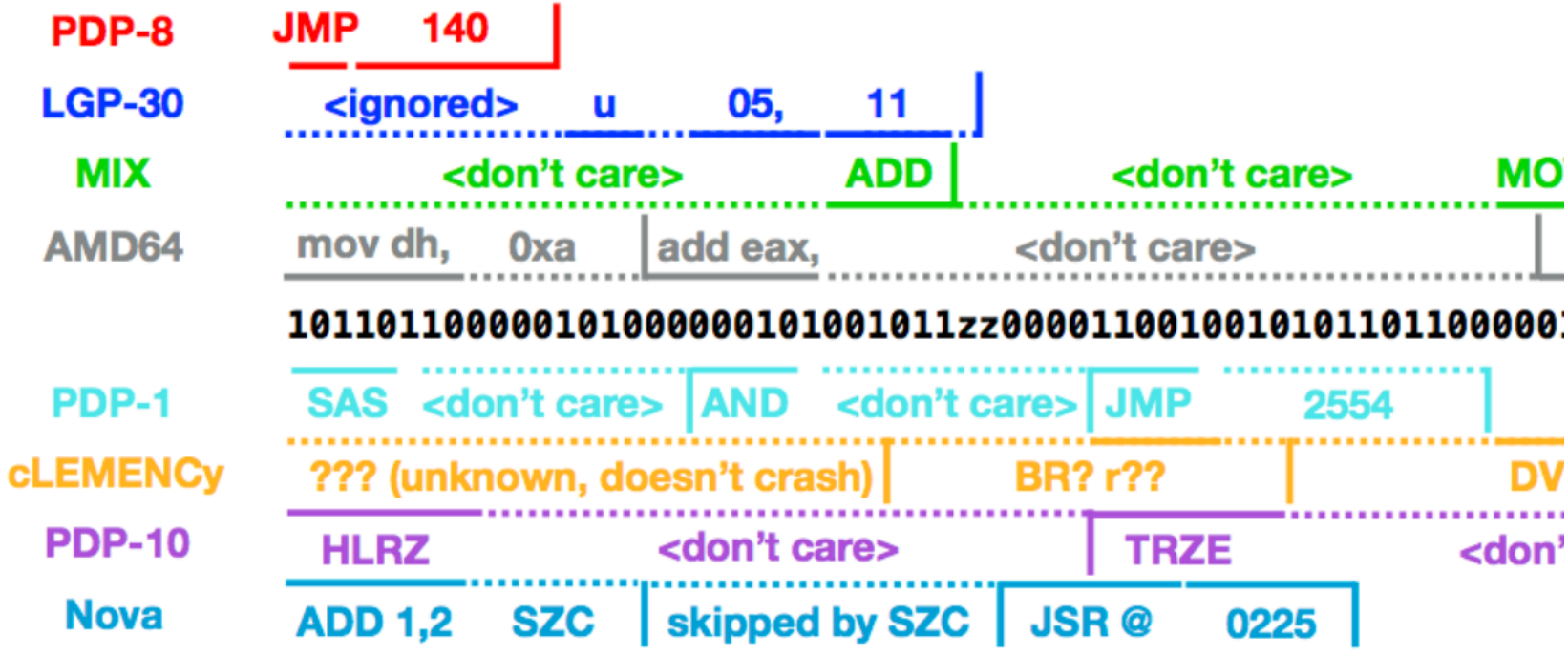
Dispatching routine

- `jz` : Jump is taken if Zero Flag (ZF) is 0.

```
1   // x86          xor eax, eax
2   // x86_64       xor eax, eax
3   xorl %eax, %eax
4
5   // x86          inc eax    ; eax = 1
6   // x86_64       REX + nop  ; eax = 0
7   .byte 0x40
8   nop
9   jz x86_64
10  <x86 shellcode>
11
12 x86_64:
13  <x86_64 shellcode>
```

DEFCON18 CTF Doublethink (12 archs!)

- Ref. https://www.robertxiao.ca/hacking/defcon2018-assembly-polyglot/



Discussion 4 (shellcode min)

1. What's your general idea?

Discussion 4 (shellcode min)

1. What's your general idea?

- Staging shellcode (env/stack)
- Use existing "/proc/flag" in the binary
- Leverage the context as much as possible (`rax`, `rsi`)?
- `fgets()` v.s., `strlen()`

Discussion 5 (shellcode ascii)

- Only use 0x20-0x7e (alphanumeric chars)
- Basic idea: construct the real shellcode on the stack at runtime

```
1  movw $0,%ax           ; 0x66 0xb8 0x00 0x00 (not allowed)
2  ->
3  andw $0x454e,%ax      ; 0x66 0x25 0x4e 0x45
4  andw $0x3a31,%ax      ; 0x66 0x25 0x31 0x3a
5
6  push $0xdead          ; 0x68 0xad 0xde 0x00 0x00 (not
lowed)
7  ->
8  subw %ax,$0x7e7e      ; 0x66 0x2d 0x7e 0x7e
9  subw %ax,$0x7e6e      ; 0x66 0x2d 0x6e 0x7e
10 subw %ax,$0x2467      ; 0x66 0x2d 0x67 0x24
11 push %ax              ; 0x66 0x50
12
13 >>> hex(0x20000 - 0x7e7e - 0x7e6e - 0x2467) = 0xdead
```

Lab03: Stack Overflow (Two Weeks)

- Finally! It's time to write real exploits (i.e., control hijacking)
- TONS of interesting challenges!
  - e.g., lack-of-four, frobnicated, upside-down ..

Lab03: Stack Overflow ('1996)!

.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One
aleph1@underground.org

`smash the stack` [C programming] n. On many C implementations
it is possible to corrupt the execution stack by writing past
the end of an array declared auto in a routine. Code that does
this is said to smash the stack, and can cause return from the
routine to jump to a random address. This can produce some of
the most insidious data-dependent bugs known to mankind.
Variants include trash the stack, scribble the stack, mangle
the stack; the term mung the stack is not used, as this is
never done intentionally. See spam; see also alias bug,

Today's Tutorial

- Example: hijacking crackme0x00!
- A template exploit code
- In-class tutorial
  - Your first stack overflow!
  - Extending the exploit template (python)

DEMO: Ghidra/crackme0x00

- Ghidra w/ crackme0x00
- Exploit writing

crackme0x00


```
$ objdump -M intel-mnemonic -d crackme0x00
...
0804869d <start>:
804869d: 55                             push   ebp
804869e: 89 e5                          mov    ebp,esp
80486a0: 83 ec 18                       sub    esp,0x18
80486a3: 83 ec 0c                       sub    esp,0xc
...

           |<=- -0x18-=>|+--- ebp
top                     v
[          [buf ..  ]   ][fp][ra]
|<=---  0x18+0xc -----=>|
```

crackme0x00


```
$ objdump -M intel-mnemonic -d crackme0x00
...
80486c6: 8d 45 e8                lea    eax,[ebp-0x18]
80486c9: 50                       push   eax
80486ca: 68 31 88 04 08           push   0x8048831
80486cf: e8 ac fd ff ff           call   8048480 <scanf@plt>

           |<=- -0x18-=>|+--- ebp
top                     v
[          [~~~~>   ]   ][fp][ra]
|<=---  0x18+0xc -----=>|
           [*****************XXXX]
```

crackme0x00

- How can we bypass the password check w/o putting the correct password?
  - Where to jmp? (i.e., where the IP should point to?)
  - How to inject a shellcode (later)?

In-class Tutorial

- Step 1: Navigate the binary with your Ghidra!
- Step 2: Play with your first exploit!
- Step 3: Using an exploit template!

```
$ ssh lab03@54.88.195.85
Password: xxxxxxx

$ cd tut03-stackovfl
$ cat README
```

References

- Phrack #49-14