Review 01: Stack Overflow Exploits and Defenses

Chengyu Song

Slides based on CS6265 taught by Prof. Taesoo Kim
Scoreboard
Course Grading (Expectation for A/B)

1. Game:
   - 40% → A
   - 30-40% → B

2. Expectation:
   - 7 on average → A
   - 5 on average → B

3. Please don't give up! I'll try my best to help you succeed!
Course Grading (Current Status)

- Lab01: 8:10, 1:7, 1:4, ~9.1
- Lab02: 3:10, 2:9, 2:8, 1:6, 1:5, 1:2, ~7.7
- Lab03: 2:10, 1:9, 1:8, 2:1

Survey: less challenges? basic + bonus challenges?
What have we covered 1

- Lab01: basic reverse engineering techniques
  - Static analysis (objdump/disas)
  - Dynamic analysis (gdb/breakpoint/examine memory)
  - Patching (nop/skip)
What have we covered 2

- Lab02: how to write shellcode
  - Write assembly: 32-bit and 64-bit
  - Transform assembly: avoid special characters
- Lab02: advanced techniques
  - Reuse existing code/data
  - Architecture detection
  - Dynamic code modification/generation
What have we covered 3

• Lab03: stack buffer overflow
  1. What can be overwritten to launch attacks?
     • return address
     • register spills
  2. Where can we "jump" to?
     • different variables on stack
     • existing code
Discussion 1

• Why we can write those exploits?

$ ../../../bin/checksec.sh --file target
RELRO     STACK CANARY    NX     PIE     RPATH     RUNPATH
Partial RELRO  No canary found  NX disabled  No PIE  No RPATH  No RUNPATH

$ sysctl kernel.randomize_va_space
kernel.randomize_va_space = 0
Discussion 2

1. **Vulnerability**
2. Ability to overwrite sensitive data
3. Ability to execute shellcode
4. Ability to know where to jump/write to
Discussion 3

• How can we stop the exploits you write?

1. Eliminate the vulnerability
2. Eliminate abilities
Stack Protection 1

- crackme0x00: the vulnerability

$ objdump -d crackme0x00
...
8048448:    8d 45 e8       lea    -0x18(%ebp),%eax
804844b:    89 44 24 04   mov     %eax,0x4(%esp)
804844f:    c7 04 24 8c 85 04 08 movl    $0x804858c,(%esp)
8048456:    e8 d5 fe ff ff  call    8048330 <scanf@plt>
...

|$<-- 0x18-->|+---- ebp
   top      v
|[    [~~~~>   ] ][fp][ra]
|<--- 0x28 ------->|
Stack Protection 2

- crackme0x00: the exploit

```
|<-- 0x18-->|+-- ebp
  top                        v
[ [~~~~> ] ][fp][ra]
|<---- 0x28        ----->|

AAAABBBB....GGGGHHHHH
```
Stack Protection 3

- Canary

```
|<-- 0x14 -------------->|+--- ebp

v

[ [ ] [canary][fp][ra][ ....]
|<---- 0x30 ------------------>

XOXOX0 XXXX
(corrupted?)
```
Stack Protection 4

• What is the core observations behind stack canary?
  1. Sequential overwritten
  2. return address and frame pointer is important

• How to people get those observations?
  • Learn from existing attacks
Stack Protection 5

• Are these two observations essential to attacks?
  1. Sequential overwritten
  2. return address and frame pointer is important

• Other subtle design questions
  • Where to put the canary? (e.g., right above ra? fp? local vars?)
  • Which value should I use as a canary? (e.g., secrete? random? per exec? per func?)
Stack Protection 6

• Lab04: Exploiting Weakness of Canary
Non-executable Data 1

- Assume stack protection can be bypassed, what else can we do?
- Ability two: execute shellcode
Non-executable Data 2

• How can we make shellcode not executable?

  • *Observation: shellcode are data*, we can just make data not executable
  • DEP: data execution prevention
  • NX/XN-bit: non-execute/execute-not bit
Non-executable Data 3

• Reminder: how to we get that observation?
  • Again, from **existing attacks**

• Question: is this essential to shellcode?
Non-executable Data 4

• Lab05: Bypassing DEP
Address Space Layout Randomization 1

• Assume the above two protections can be bypassed, what else can we do?
• Ability three: know where to jump/write to
Address Space Layout Randomization 2

- How can we prevent attackers from knowing the addresses?
  - Let's randomize the locations of code and data
  - Address Space Layout Randomization (ASLR)
Address Space Layout Randomization 3

- What are the assumptions of ASLR?
  1. Attackers must know where to jump to
  2. Attackers cannot predict/guess the layout
  3. Attackers cannot leak the randomness
- Question: do the above assumptions always hold?
Address Space Layout Randomization 4

- Lab06: Bypass ASLR
Take Aways

1. Lab04 - Lab06: yes, they are bypassable, but also makes the attack a lot harder

2. Are learning from existing attacks a good strategy?
By the way, how to fix crackme0x00's bug?

- scanf("%s", buf)
+ scanf("%15s", buf);