Software Security II: Memory Errors - Attacks & Defenses

Chengyu Song

Slides modified from Dawn Song
Administrivia

- Lab1
  - Writeup
Buffer overflow

- Out-of-bound memory writes (mostly sequential)
- Allow attackers to overwrite critical data (e.g., return address) to hijacking control flow to execute arbitrary code
Another arm race

- Defenders: how can we prevent the attack?
- Attackers: how can we bypass the defense?
Recall: stack buffer overflow

/* example2.c */
void function(char *str) {
    char buffer[16];
    strcpy(buffer, str);
}

void main() {
    char large_string[256];
    int i;
    for (i = 0; i < 255; i++)
        large_string[i] = 'A';
    function(large_string);
}
How can we prevent the attack?

1. Fix the root cause (best option but not always doable)
   - Why? Delays, performance, compatibility, etc

2. Prevent the exploit
Fix stack buffer overflow

- What causes the overflow?
  - The source buffer is too large
  - The destination buffer is too small
  - Forget to check size before copying

- Which one would you choose? Why?
Safer string operations

- `strcat`, `strcpy`, `sprintf`, ... are **DANGEROUS**
  - Compiler would warn you for using them
- Safer version: `strncat`, `strncpy`, `snprintf`
  - Safer but always?
  - What does `n` mean? # of characters to be copied
    - How to make sure there's enough space left?
    - What if `n` is larger than `strlen(src)`?
  - Null-terminator?
Safer string operations (cont.)

- BSD: `strlcat`, `strlcpy`, `slprintf`
  - Copy \( n - 1 \), always add '\0'
- Windows: `strncat_s`, `strncpy_s`, `snprintf_s`
  - Copy \( \min(n, \text{strlen(src)}) \)
  - Abort if `size(dest)` is not enough
  - No padding
Take away (1)

• Patching solves the root cause but
  • Requires time to develop
  • Relies on developers
  • May be wrong

• Q: is there alternative ways that do not require efforts from developers?
  • Generic mitigation techniques
Recall: stack buffer overflow exploit

<table>
<thead>
<tr>
<th>bottom of memory</th>
<th>sfp</th>
<th>ret</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>top of memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>^</td>
<td>^</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>^</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| |_____________||____________| (1)
| (2) |_____________||
| |_____________| (3)

11
What are the key steps?

1. Overwrite the return address, sequentially
2. Jump to the beginning of the shellcode
3. Execute the shellcode
Idea 1: stack canary

- Check if the return address has been corrupted before return, but how?
- How about insert a **canary** between the **return address** and **local variables**
  - Would this work? Why?

```
stack top
[ buffer ][sfp][canary][ra][args ....]
```
Not that simple!

• Which value should I use as a canary?
  • secrete? random? randomize per exec? per func?
• Where to put the canary?
  • Just protect RA? What about FP and other local variables?
• How to compare the canary value?
  • Compare? Encoding (xor)?
• What to do after you find the canary value is corrupted?
  • Crash? Report?
Take away (2)

- Stack canary makes exploit much harder
  - GCC: \texttt{-fstack-protector(-strong|full)}
  - MSVC: \texttt{/GS}
  - Random value, per execution, both RA and FP, check and report
- But it's not perfect and can be bypassed
Idea 2: non-executable data

- Observation: injected shellcode is data, why data should be executable?
- Let's make data not executable
  - Software-based approach: W^X, DEP (early stage)
  - Hardware-based approach: NX (x86), XN (ARM)
- Huge success - code injection is almost extinguished
  - Why? Very low performance overhead yet extreme effective
Countermeasures

• Idea: if I cannot inject code, can I reuse existing code?
  • **Code Reuse Attacks** (CRA)
  • Whole function reuse (e.g., `system`, `mprotect`, `mmap`)
  • Partial reuse: **Return-oriented Programming** (ROP)
    • Chain small code snippets
Take away (3)

- Defense mechanism should eliminate the key prerequisite of attacks
  - Effectiveness
- Hardware assistant can reduce a lot of overhead
  - Performance
- However, since the root cause is not eliminated, DEP can still be bypassed
Idea3: where is the payload?

- Similar to stack cookie, can we randomize the location of memory so it will be very difficult to locate the payload (shellcode, code gadgets)
  - **Address Space Layout Randomization (ASLR)**
How does ASLR work?

- **Linux**
  - Randomize the base of mmap, stack, and heap (brk)
  - Executables are loaded by mmap so their location is also randomized

- **Windows**
  - Before Windows 8, similar
  - High entropy ASLR, check references
ASLR weakness (1)

- Entropy, entropy, entropy!
- Without enough "randomness", attackers can just guess
- Two attack strategies
  - Brute-force → hacking blind
  - Spray
ASLR weakness (2)

- Predictable
  - Not fine-grained: relative offset is not changed
  - Legacy, not randomizable/randomized content
- Information leak
  - Memory disclosure
  - Side-channels
Take away (4)

- Randomization is a good (low overhead) defense strategy
  - Stack canary, ASLR, etc
- ONLY IF
  - There's enough **entropy**
  - There's **no information leak**
Summary

- Best practice to prevent buffer overflow
  - Safe programming languages: Java, Rust, Go, etc.
  - Secure coding practices: safer string operations, etc
- Three widely deployed exploit prevention techniques
  - Stack canary (cookie/guard)
  - DEP (NX/XN)
  - ASLR
Questions

• Besides missing bound check, any other bugs can also cause out-of-bound access?
• Besides return address (frame pointer), any other types of data can be overwritten to launch attacks?
For next class ...

- Software Security III: memory errors