Binary analysis: Defense Mechanism

Yue Duan
Outline

- Control-flow problems
- Simple defense mechanisms
- Research paper:
  - Control-Flow Integrity
Control-flow Violation

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read from stdio (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

<table>
<thead>
<tr>
<th>Buffer (8 bytes)</th>
<th>Overflow (2 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASSWORD</td>
<td>1 2</td>
</tr>
</tbody>
</table>
Control-flow Violation
Control-flow Violation

- malicious code (shellcode)
- compromised address
- base pointer
- buffer

- malicious code (shellcode)
- compromised address
- base pointer
- buffer

- nop
- nop
- ... multiple nops
- nop

- nop
- nop
- ... multiple nops
- nop
Shellcode

```c
#include <stdio.h>

int main() {
    char *name[2];

    name[0] = "\"/bin/sh\"\";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

<table>
<thead>
<tr>
<th>Line</th>
<th>Assembly Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xorl %eax, %eax</td>
<td># push 0 into stack (end of string)</td>
</tr>
<tr>
<td>2</td>
<td>pushl %eax</td>
<td># push &quot;/bin/sh&quot; into stack</td>
</tr>
<tr>
<td>3</td>
<td>pushl $0x68732f2f</td>
<td># push &quot;/bin&quot; into stack</td>
</tr>
<tr>
<td>4</td>
<td>pushl $0x6962622f</td>
<td># push &quot;/sh&quot; into stack</td>
</tr>
<tr>
<td>5</td>
<td>movl %esp, %ebx</td>
<td># %ebx = name[0]</td>
</tr>
<tr>
<td>6</td>
<td>pushl %eax</td>
<td># name[1]</td>
</tr>
<tr>
<td>7</td>
<td>pushl %ebx</td>
<td># name[0]</td>
</tr>
<tr>
<td>8</td>
<td>movl %esp, %ecx</td>
<td># %ecx = name</td>
</tr>
<tr>
<td>9</td>
<td>cdq</td>
<td># %edx = 0</td>
</tr>
<tr>
<td>10</td>
<td>movb $0x0b, %al</td>
<td># invoke execve(name[0], name, 0)</td>
</tr>
<tr>
<td>11</td>
<td>int $0x80</td>
<td></td>
</tr>
</tbody>
</table>
Steps

● insert shellcode into somewhere in memory
● overwrite return address with shellcode’s address
● when function returns, shellcode will be executed
● a shell will be launched
● if the vulnerable program happens to be root-owned SetUID program
Fundamental Problems

- No distinction between code and data
  - data in the process can be interpreted as code
- Attacker can easily redirect control flow to the injected data (code)
Simple Defense Mechanisms

- Data Execution Prevention (DEP)
- Address Space Layout Randomization (ASLR)
- Stack Canary
Data Execution Prevention

- System-level memory protection feature
- DEP prevents code from being run from data pages
  - heap, stacks, etc
- If an application attempts to run
  - memory access violation exception is raised
  - process is terminated
Data Execution Prevention
Data Execution Prevention

malicious code (shellcode)

compromised address

base pointer

buffer

not executable

SEGFAULT
Evasion

- What if ...
  - attacker does not inject code
  - instead, he reuses existing code

Return-to-libc attack: reuse code in Libc
Return-to-Libc

- Malicious code (shellcode)
- Attacker knows the address of Libc
- Based pointer
- Buffer
- Libc.so
- System()
Address Space Layout Randomization (ASLR)

- Multiple attacks rely on guessing addresses
  - inject shellcode
  - return-to-libc

- Key idea:
  - Can we make the guessing practically impossible?
Address Space Layout Randomization (ASLR)

- Randomize the address space positions
  - base of the executable
  - stack
  - heap
  - libraries

- Possible evasions
  - bruteforce
  - information leakage
  - return-oriented programming (ROP)

(a) ASLR disabled
(b) ASLR enabled
Return-Oriented Programming (ROP)

- Key idea:
  - reuse existing code without knowing the exact addresses
  - find meaningful code ‘gadgets’
  - chain ‘gadgets’ together with return to complete a malicious action

```
xor eax, eax
ret

pop ebx
pop eax
ret

add eax, ebx
ret
```
Return-Oriented Programming (ROP)
Stack Canary

- Also called StackGuard
- random integer
  - next to the return address
- before return, always check if
  - canary has been changed
- Built-in feature for most of the modern compilers
- Can be guessed
Control-Flow Integrity

M Abadi, M Budiu, Ú Erlingsson, J Ligatti

ACM CCS 2005
Motivation

● Many attacks involve control-flow hijacking
● Existing defense mechanisms
  ○ either impractical
  ○ or need hardware support
● Fundamental limitations
  ○ lack of a realistic attack model
  ○ reliance on informal reasoning and hidden assumptions

● Major challenge: How to protect control-flow from being hijacked?
Control-Flow Integrity

● Key idea:
  ○ execution should always follow the pre-defined control flow (CFI security policy)!

● CFI policy
  ○ extracted from Control-flow graph (CFG)
  ○ enforced at runtime with checks
Example

```c
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}
```
CFI Enforcement

- For each control transfer, such as a function call
  - statically determine its possible destinations
- Insert a **unique bit pattern** at every destination
  - two destinations are considered equivalent if CFG contains edges to each from the same source
- Insert codes that
  - enforce runtime check
  - whether the bit pattern of the target instruction matches the pattern of possible destinations
- Done by binary instrumentation technique
CFI Enforcement - jmp

original code

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
</tr>
</tbody>
</table>

; computed jump

instrumentation (a)

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>81 39 78 56 34 12</td>
<td>cmp [ecx], 12345678h</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
</tr>
<tr>
<td>8D 49 04</td>
<td>lea ecx, [ecx+4]</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
</tr>
</tbody>
</table>

; comp ID & dst
; if != fail
; skip ID at dst
; jump to dst

instrumentation (b)

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 77 56 34 12</td>
<td>mov eax, 12345677h</td>
</tr>
<tr>
<td>40</td>
<td>inc eax</td>
</tr>
<tr>
<td>39 41 04</td>
<td>cmp [ecx+4], eax</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
</tr>
</tbody>
</table>

; load ID-1
; add 1 for ID
; compare w/dst
; if != fail
; jump to label

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E 0F 18 05</td>
<td>prefetchnta</td>
</tr>
<tr>
<td>78 56 34 12</td>
<td>[12345678h]</td>
</tr>
<tr>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
</tr>
</tbody>
</table>

; label
; ID
; ID
; dst
; dst
CFI Enforcement - ret

original code

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 53 08</td>
<td>call [ebx+8]</td>
<td>C2 10 00</td>
<td>ret 10h</td>
</tr>
<tr>
<td></td>
<td>; call fptr</td>
<td></td>
<td>; return</td>
</tr>
</tbody>
</table>

instrumentation

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Instructions</th>
<th>Opcode bytes</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>8B 43 08</td>
<td>mov eax, [ebx+8]</td>
<td>8B 0C 24</td>
<td>mov ecx, [esp]</td>
</tr>
<tr>
<td>3E 81 78 04</td>
<td>cmp [eax+4], 12345678h</td>
<td>83 C4 14</td>
<td>add esp, 14h</td>
</tr>
<tr>
<td>78 56 34 12</td>
<td></td>
<td>3E 81 79 04</td>
<td>cmp [ecx+4],</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td>DD CC BB AA</td>
<td>AABBCDDDh</td>
</tr>
<tr>
<td>FF D0</td>
<td>call eax</td>
<td></td>
<td>w/ID</td>
</tr>
<tr>
<td>3E 0F 18 05</td>
<td></td>
<td>75 13</td>
<td>jne error_label</td>
</tr>
<tr>
<td>DD CC BB AA</td>
<td></td>
<td>FF E1</td>
<td>jmp ecx</td>
</tr>
<tr>
<td>AA</td>
<td>prefetchnta [AABBCDDDh] ; label ID</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CFI Precision

• Assume that:
  ○ A() calls C()
  ○ B() calls C() or D() (When can this happen?)
• CFI will use the same tag for C and D
  ○ allow A to call D
• Possible solutions:
  ○ duplicate code or inlining
  ○ multiple tags
CFI Precision

- function F() is called twice from A() and B()
  - CFI uses the same tag for both call sites
  - allow F() to return to B() after being called from A()
- Solution: shadow call stack
  - always guarantee the return to the latest call site
Evaluation: performance overhead

- modest performance overhead
  - on average 21%
  - 5% for gzip and 11% for gcc

Figure 8: Enforcement overhead for CFI with a protected shadow call stack on SPEC2000 benchmarks.
Thank you!

Question?