Advanced Systems Security: Control-Flow Integrity

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Quiz Questions

1 – Why randomize code layout with ASLR (one reason)?

2 – Why randomize data layout with ASLR given DEP (one reason)?

3 – What are indirect memory disclosures?
Vulnerability

• How do you define computer ‘vulnerability’?
Buffer Overflow

• First and most common way to take control of a process

• Attack code
  ‣ Call the victim with inputs necessary to overflow buffer
  ‣ Overwrites the return address on the stack

• Exploit
  ‣ Jump to attacker chosen code
  ‣ Run that code
Determine what to attack

- Local variable that is a char buffer
  - Called buf

```c
... printf("BEFORE picture of stack\n");
for ( i=((unsigned) buf-8); i<((unsigned) ((char *)&ct)+8); i++ )
  printf("%p: 0x%x\n", (void *)i, *(unsigned char *) i);
/* run overflow */
for ( i=1; i<tmp; i++ )
  printf("i = %d; tmp= %d; ct = %d; &tmp = %p\n", i, tmp, ct, (void *)&tmp);
  strcpy(p, inputs[i]);
/* print stack after the fact */
printf("AFTER iteration %d\n", i);
for ( j=((unsigned) buf-8); j<((unsigned) ((char *)&ct)+8); j++ )
  printf("%p: 0x%x\n", (void *)j, *(unsigned char *) j);
p += strlen(inputs[i]);
if ( i+1 != tmp )
  "p++ = '\';
}
printf("buf = %s\n", buf);
printf("victim: %p\n", (void *)&victim);
return 0;
```
Configure Attack

- Configure following
  - Distance to return address from buffer
    - Where to write?
  - Location of start of attacker’s code
    - Where to take control?
  - What to write on stack
    - How to invoke code (jump-to existing function)?
  - How to launch the attack
    - How to send the malicious buffer to the victim?
Return Address

- x86 Architecture
  - Build 32-bit code for Linux environment
- Remember integers are represented in “little endian” format
- Take address 0x8048471
  - See trace at right

```c
// BEFORE picture of stack
0xbfa3b854: 0x3
0xbfa3b855: 0x0
0xbfa3b856: 0x0
0xbfa3b857: 0x0
0xbfa3b858: 0x3
0xbfa3b859: 0x0
0xbfa3b85a: 0x0
0xbfa3b85b: 0x0
0xbfa3b85c: 0x0
0xbfa3b85d: 0x0
0xbfa3b85e: 0x0
0xbfa3b85f: 0x0
0xbfa3b860: 0x0
0xbfa3b861: 0x0
0xbfa3b862: 0x0
0xbfa3b863: 0x0
0xbfa3b864: 0x0
0xbfa3b865: 0x0
0xbfa3b866: 0x0
0xbfa3b867: 0x0
0xbfa3b868: 0xa8
0xbfa3b869: 0xb8
0xbfa3b86a: 0xa3
0xbfa3b86b: 0xbf
0xbfa3b86c: 0x71
0xbfa3b86d: 0x84
0xbfa3b86e: 0x4
0xbfa3b86f: 0x8
0xbfa3b870: 0x3
0xbfa3b871: 0x0
0xbfa3b872: 0x0
0xbfa3b873: 0x0

// buf

// ebp

// rtn addr

// ct
```
Anatomy of Control Flow Attacks

• Two steps

• First, the attacker changes the control flow of the program
  ‣ In buffer overflow, overwrite the return address on the stack
  ‣ What are the ways that this can be done?

• Second, the attacker uses this change to run code of their choice
  ‣ In buffer overflow, inject code on stack
  ‣ What are the ways that this can be done?
Return-oriented Programming

• General approach to control flow attacks

• Demonstrates how general the two steps of a control flow attack can be

• First, change program control flow
  ‣ In any way

• Then, run any code of attackers’ choosing - code in the existing program
  ‣ From starting address (gadget) to ret
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code

```
%eax = 0x8048000 =
%ebx =

pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret
```

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop %eax</td>
<td>G1 5</td>
</tr>
<tr>
<td>ret</td>
<td>G2 0x8048000</td>
</tr>
<tr>
<td>pop %ebx</td>
<td>G3 ...</td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>movl %eax, (%ebx)</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td></td>
</tr>
</tbody>
</table>

Registers
%eax =
%ebx =

Memory
0x8048000 =

Return Address
buf
• Use ESP as program counter
  
  ‣ E.g., Store 5 at address 0x8048000
  
  • without introducing new code

**Code**

G1: pop %eax
    ret

G2: pop %ebx
    ret

G3: movl %eax, (%ebx)
    ret

**Stack**

G1
5
G2
0x8048000
G3
...

**Memory**

0x8048000 =

**Registers**

%eax =
%ebx =
• Use ESP as program counter
  ▸ E.g., Store 5 at address 0x8048000
  ▸ without introducing new code

```c
%eax =
%ebx =
0x8048000 =
```

**Code**

G1: pop %eax
ret

G2: pop %ebx
ret

G3: movl %eax, (%ebx)
ret

**Stack**

Return Address

buf

G1
5
G2
0x8048000
G3
...

**Memory**

0x8048000 =
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code

```
%eax = 5
%ebx =
G1: pop %eax
    ret
G2: pop %ebx
    ret
G3: movl %eax, (%ebx)
    ret

%eax = 5
%ebx =

G1
    5
G2
    G2
0x8048000
    0x8048000
G3
    ...
```

Code

Stack

Registers

Memory

Return Address

buf

```
```
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: pop %eax ret</td>
<td>G1</td>
</tr>
<tr>
<td>G2: pop %ebx ret</td>
<td>5</td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx) ret</td>
<td>G2 0x8048000 G3 ...</td>
</tr>
</tbody>
</table>

Registers
%eax = 5
%ebx = 0x8048000
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
• without introducing new code

```
%eax = 5
%ebx = 0x8048000
```

```
G1: pop %eax
    ret

G2: pop %ebx
    ret

G3: movl %eax, (%ebx)
    ret
```

```
<table>
<thead>
<tr>
<th>Stack</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>G3</td>
</tr>
<tr>
<td>5</td>
<td>0x8048000 =</td>
</tr>
<tr>
<td>G2</td>
<td></td>
</tr>
<tr>
<td>0x8048000</td>
<td></td>
</tr>
<tr>
<td>G3</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```

Return Address

buf
• **Use ESP as program counter**
  - E.g., Store 5 at address \(0x8048000\)
  - without introducing new code

```plaintext
%eax = 5
%ebx = 0x8048000
```

**Code**

G1: pop %eax
    ret

G2: pop %ebx
    ret

G3: movl %eax, (%ebx)
    ret

**Stack**

<table>
<thead>
<tr>
<th>G1</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>G2</td>
<td>0x8048000</td>
</tr>
<tr>
<td>G3</td>
<td>...</td>
</tr>
</tbody>
</table>

**Return Address**

buf

**Memory**

0x8048000 = 5
Prevent ROP Attacks

• How would you prevent a program from executing gadgets rather than the expected code?
Prevent ROP Attacks

• How would you prevent a program from executing gadgets rather than the expected code?
  ‣ Control-flow integrity
    • Force the program to execute according to an expected CFG
Control-Flow Integrity

Our Mechanism

- \( F_A \):
  - \( \text{call } fp \)
  - \( \text{nop IMM}_2 \)
  - if(\(*fp \neq \text{nop IMM}_1\)) halt

- \( F_B \): nop IMM_1

- \( \text{call+1} \)

- return

\[ \text{if(\(**esp \neq \text{nop IMM}_2\)) halt} \]

NB: Need to ensure bit patterns for nops appear nowhere else in code memory

CFG excerpt

- \( A_{\text{call}} \rightarrow B_1 \)
- \( A_{\text{call}+1} \leftarrow B_{\text{ret}} \)
Control-Flow Integrity

More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int → int function

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

CFG excerpt

$A_{\text{call}} \xrightarrow{} B_1$
$succ(A_{\text{call}}) = \{B_1, C_1\}$

if(*fp != nop IMM) halt

$F_A$
$F_B$
$F_C$
Control-Flow Integrity

Imprecise Return Information

Q: What if \( F_B \) can return to many functions?
A: Imprecise CFG

CFG excerpt

\[
\text{succ}(B_{\text{ret}}) = \{ A_{\text{call+1}}, D_{\text{call+1}} \}
\]

CFG Integrity: Changes to the PC are only to valid successor PCs, per succ().
Destination Equivalence

• Eliminate impossible return targets
  ‣ Two destinations are said to be equivalent if connect to a common source in the CFG.

![Diagram of function calls and returns]

Figure 4. Destination equivalence effect on ret instructions (a dashed line represents an indirect call while a solid line stands for a direct call)
Destination Equivalence

- Eliminate impossible return targets
  - Can $R2$ be a return target of $\text{function}_j$?

Figure 4. Destination equivalence effect on $\text{ret}$ instructions (a dashed line represents an indirect $\text{call}$ while a solid line stands for a direct $\text{call}$)
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

$A_{call}$ \rightarrow $B_1$  
$E_{call}$ \rightarrow $C_1$

CFG excerpt

$A_{call}$ \rightarrow $B_1$  
$E_{call}$ \rightarrow $C_{1E}$

$C_{1A}$

$C_{1E}$
Restricted Pointer Indexing

- One table for call and return for each function

- Why can’t function_j return to R2 with this approach?
Control-Flow Graph

- CFI enforces an expected CFG
  - Each call-site transfers to expected instruction
  - Each return transfers back to expected call-site
- Direct calls
  - Call instructions targeted for specific instruction – no problem
- Indirect calls
  - Function pointers – what are the possible targets?
- Returns
  - Determine return target dynamically – can be overwritten
- Can we compute an accurate CFG?
Enforce CFG

- Challenge in computing an enforceable CFG
  - Targets computed dynamically, so how can we predict in advance and without generating any false positives

- Coarse-grained CFG
  - Any function is a legal indirect call target (ICT)
  - Any call-site is a legal return target

- Signature-based
  - Function with same signature as call-site is a valid ICT

- Taint-based
  - Track function symbols that can reach a ICT
• If function pointers are used in a restricted way, we can predict the indirect call targets using taint analysis
  
  Assumption 1: The only allowed operations on a function pointer variable are assignment and dereferencing (for call)
  
  Assumption 2: There exist no data pointer to a function pointer

- FreeBSD and MINIX largely follow these assumptions
Shadow Stack

- Method for maintaining return targets for each function call reliably

- On call
  - Push return address on the regular stack
  - Also, push the return address on the shadow stack

- On return
  - Validate the return address on the regular stack with the return address on the shadow stack

- Why might this work? Normal program code cannot modify the shadow stack memory directly
Other Problems with CFI

• CFI enforcement can be expensive

• Idea: only check CFI lazily
  ‣ kBouncer inspects the last 16 indirect branches taken each time the program invokes a system call
    • Why 16? Uses Intel’s Last Branch Record (LBR), which can store 16 records
  ‣ ROPecker also checks forward for future gadget sequences (short sequences ending in indirection)

• These hacks do not work – See papers in USENIX Security 2014 for attacks against

• Bottom line – no shortcuts
Control-Flow Bending

• Do we need a shadow stack?
  ‣ After applying coarse-grained CFG

<table>
<thead>
<tr>
<th></th>
<th>AIR</th>
<th>Gadget red.</th>
<th>Targets</th>
<th>Gadgets</th>
</tr>
</thead>
<tbody>
<tr>
<td>No CFI</td>
<td>0%</td>
<td>0%</td>
<td>1850580</td>
<td>128929</td>
</tr>
<tr>
<td>CFI</td>
<td>99.06%</td>
<td>98.86%</td>
<td>19611</td>
<td>1462</td>
</tr>
</tbody>
</table>

Table 1: Basic metrics for the minimal vulnerable program under no CFI and our coarse-grained CFI policy.

• Still lots of choices and gadgets
Control-Flow Bending

• Do we need a shadow stack?
  ‣ After applying precise CFG

• Problem: Dispatcher functions
  ‣ A function that can overwrite its return address when given adversary controlled input argument values
  ‣ Even with buffer overflow protection (stackguard)
  ‣ E.g., consider memcpy

• How would you use a dispatcher function to control execution while evading CFI?
Control-Flow Bending

- Do we need a shadow stack?
  - After applying precise CFG

- Problem: *Dispatcher functions*
  - A function that can overwrite its return address when given adversary controlled input argument values
  - Even with buffer overflow protection (stackguard)
  - E.g., consider memcpy

- How would you block a dispatcher function from launching an ROP?
Control-Flow Bending

- If we have a fine-grained CFG and a shadow stack are we safe from control-flow bending?
Control-Flow Bending

• If we have a fine-grained CFG and a shadow stack are we safe from control-flow bending?
• Unfortunately, no.
  ‣ Turing-complete functions
    • A function that has a memory read and memory write
    • A conditional jumps and loops
  ‣ Examples of these functions
    • printf
    • fputs
Take Away

- **Memory errors** are the classic vulnerabilities in C programs (**buffer overflow**)
  - Despite years of exploration into defenses, a Turing-complete approach to exploitation remains given an appropriate memory error (**return-oriented programming**)

- **Control-flow integrity** has been suggested as the way to block ROP attacks
  - Not as easy as it sounds
  - CFI enforcement requires a fine-grained CFG and shadow stack (or equivalent)

- Yet, still some ROP attacks are possible (**bending**)