Lec 14: CFI

CS492E: Introduction to Software Security

Sang Kil Cha
Defense Techniques So Far ...

- DEP
- ASLR
- Canary

Problem: control-flow hijacking still possible
Control Flow Hijack Exploit

Attacker’s own code
e.g., install malicious software
Can we enforce control-flow integrity?
CFI Policy

The CFI security policy dictates that software execution must follow a path of a Control-Flow Graph (CFG) determined \textit{ahead of time}. 

Quote from control flow integrity, CCS 2005
CFG (Control Flow Graph)

A CFG is a graph that represents all paths that might be traversed through a program execution.
CFG (Control Flow Graph)

Each node in a CFG represents a **basic block**

* Basic Block:
  A sequence of statements that is always entered at the beginning and exited at the end*

* Quote from Modern Compiler Implementation
### Basic Block

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>55</code></td>
<td>push ebp</td>
</tr>
<tr>
<td>1</td>
<td><code>89 e5</code></td>
<td>mov ebp,esp</td>
</tr>
<tr>
<td>3</td>
<td><code>83 ec 10</code></td>
<td>sub esp,0x10</td>
</tr>
<tr>
<td>6</td>
<td><code>c7 45 f8 00 00 00 00</code></td>
<td>mov DWORD PTR [ebp-0x8],0x0</td>
</tr>
<tr>
<td>d</td>
<td><code>c7 45 fc 0a 00 00 00</code></td>
<td>mov DWORD PTR [ebp-0x4],0xa</td>
</tr>
<tr>
<td>14</td>
<td><code>eb 08</code></td>
<td>jmp 1e &lt;v+0x1e&gt;</td>
</tr>
<tr>
<td>16</td>
<td><code>83 45 f8 01</code></td>
<td>add DWORD PTR [ebp-0x8],0x1</td>
</tr>
<tr>
<td>1a</td>
<td><code>83 6d fc 01</code></td>
<td>sub DWORD PTR [ebp-0x4],0x1</td>
</tr>
<tr>
<td>1e</td>
<td><code>83 7d fc 00</code></td>
<td>cmp DWORD PTR [ebp-0x4],0x0</td>
</tr>
<tr>
<td>22</td>
<td><code>7f f2</code></td>
<td>jg 16 &lt;v+0x16&gt;</td>
</tr>
<tr>
<td>24</td>
<td><code>8b 45 f8</code></td>
<td>mov eax,DWORD PTR [ebp-0x8]</td>
</tr>
<tr>
<td>27</td>
<td><code>c9</code></td>
<td>leave</td>
</tr>
<tr>
<td>28</td>
<td><code>c3</code></td>
<td>ret</td>
</tr>
</tbody>
</table>
CFI = Any Execution Should Follow Control Paths of This CFG
CFI Assumptions

- Attackers cannot execute data (DEP is enabled)
- Programs cannot change themselves (no self-modifying code)
How to Enforce CFI?

• Give *unique* IDs at destinations
• For all branch instructions, check destination IDs before taking the branch
How to Instrument?

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Destination Opcode bytes</th>
<th>Destination 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] ; dst</td>
</tr>
<tr>
<td></td>
<td>; computed jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>81 39 78 56 34 12</td>
<td>cmp [ecx], 12345678h</td>
<td>78 56 34 12</td>
<td>; data 12345678h ; ID</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4] ; dst</td>
</tr>
<tr>
<td>8D 49 04</td>
<td>lea ecx, [ecx+4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

or, alternatively, instrumented as (b):

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Destination Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 77 56 34 12</td>
<td>mov eax, 12345677h</td>
<td>3E 0F 18 05</td>
<td>prefetchnta ; label</td>
</tr>
<tr>
<td>40</td>
<td>inc eax</td>
<td>78 56 34 12</td>
<td>[12345678h] ; ID</td>
</tr>
<tr>
<td>39 41 04</td>
<td>cmp [ecx+4], eax</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4] ; dst</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Can be instrumented as (a):
CFI Challenge

What if a single branch instruction can jump to multiple addresses? (e.g., call eax)
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);
}
```

Image from control flow integrity, CCS 2005
Can you spot labeling problems?

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

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

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

Image from control flow integrity, CCS 2005
Problem: What if D returns to A?
Potential Solutions

• Multiple tags
• Shadow call stack

What’s the problem?
Another Problem

Context insensitive!

A → B
A → C

B Label 42
C Label 42
Shadow Call Stack

• In function prologues, store the return address in another area of memory

• In function epilogues, check if we are returning to the proper address

A Binary Rewriting Defense against Stack based Buffer Overflow Attacks, USENIX ATC 2003
CFI with Shadow Call Stack

```
call eax ; call func ptr                             ret ; return

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

add gs:[0h], 4h ; inc stack by 4
mov ecx, gs:[0h] ; get top offset
mov gs:[ecx], LRET ; push ret dst
cmp [eax+4], ID ; comp fpTR w/ID
jne error_label ; if != fail
call eax ; call func ptr

mov ecx, gs:[0h] ; get top offset
mov ecx, gs:[ecx] ; pop return dst
sub gs:[0h], 4h ; dec stack by 4
add esp, 4h ; skip extra ret
cmp ecx, 0 ; check if return
jne ret_label ; jump to return
```
Time of Check to Time of Use

```c
if (access("file", W_OK) != 0) {
    exit(1); // exit if not writable
}

fd = open("file", O_WRONLY);
write(fd, buffer, sizeof(buffer));
```

TOCTTOU

call eax ; call func ptr

ret ; return

with a CFI-based implementation of a protected shadow call stack using hardware segments, can become:

add gs:[0h], 4h ; inc stack by 4
mov ecx, gs:[0h] ; get top offset
mov gs:[ecx], LRET ; push ret dst
cmp [eax+4], ID ; comp fpTR w/ID
jne error_label ; if != fail
call eax ; call func ptr

mov ecx, gs:[0h] ; get top offset
mov ecx, gs:[ecx] ; pop return dst
sub gs:[0h], 4h ; dec stack by 4
add esp, 4h ; skip extra ret
jmp ecx ; jump return dst

LRET: ...

TOCTTOU can happen here if ret is used

Image from control flow integrity, CCS 2005
Runtime Overhead

Image from control flow integrity, CCS 2005
CFI Practical Implication?

• CFI on binary code is difficult
  - Subtlety of Vulcan

• CFI is slow
CFI on Binary: Legacy Code

- CFG reconstruction from binary is difficult
- Indirect jumps?
CFI on Binary: Bypassing CFI

• Dynamically generated code
  – Self modifying code (e.g., packing)
  – JIT compiled code

• CFI is not perfect anyways
CFI Practicality: Coarse-Grained CFI

• Practical Control Flow Integrity and Randomization for Binary Executables, *Oakland 2013*

• Control Flow Integrity for COTS binaries, *USENIX Security 2013*

• Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, *USENIX Security 2013*

• ROPecker: A Generic and Practical Approach for Defending against ROP attacks, *NDSS 2014*
CFI Practicality: Coarse-Grained CFI

- Reduce the # of labels to check (e.g., checks if a function returns to a call-preceded instruction)
- Employ behavioral heuristics to quickly check integrity (e.g., detect gadget-like sequences)
Attacking Coarse-Grained CFI

• Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, *USENIX Security 2014*

• Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard, *USENIX Security 2014*

• Out of Control: Overcoming Control-Flow Integrity, *Oakland 2014*
CFI is Now in Major Compilers

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, *USENIX Security 2014*

Protect forward edges with
- VTV (VTable Verification)
- IFCC (Indirect Function Call Checker)
- FSAN (Indirect Function Call Sanitizer)
Performance vs. Security

Still not solved 😞
Implication of Shadow Call Stack

What if we have a perfect CFI, but without shadow call stack?

We can return to some functions that are not in the CFG
CFI Without Shadow Call Stack

• ROP may be possible, but not easy

• Return-into-libc is much easier though
  – system calls memcpy
  – If a vulnerable function can call memcpy, then we can jump back to system (with a dispatcher function)

Dispatcher Function

A function that can overwrite its own return address when given arguments supplied by an attacker.

Any function that has a “write-what-where” primitive

E.g. memcpy, printf, fputs, etc.
memcpy

memcpy(dst, src, 8);

We can jump to any caller of memcpy
Eval: CFI Without Shadow Call Stack

• Analyzed 6 apps.

• Successfully exploited 5 apps. assuming fully precise static CFI without shadow call stack
What about Fully Precise CFI?

• We now assume we use shadow call stack

• We cannot use dispatcher functions any more

• Are we secure now?
Printf-Oriented Programming

• A single call to printf allows an attacker to perform Turing-complete computation!

• Assume we can fully control the arguments to printf

• Can bypass fully precise CFI
Printf-Oriented Programming

- Memory read: %s
- Memory write: %n
- Conditional?
Conditional

if ( *c ) {
    *t = x;
}

Single byte write that overwrite Q
If NULL byte is written, printf terminates

Address of Q

"%s%hhnQ%*d%n", c, s, x-2, 0, t

Width specifier
Turing Complete!

Image from the slides of Control-Flow Bending: On the Effectiveness of Control-Flow Integrity, USENIX Security 2015
Printf-Oriented Programming

- Single call to printf is enough to run any arbitrary code
- No need to violate CFI
Question

Do you think printf-oriented-programming-based attacks hijack control flow?
Questions?