Control-Flow Integrity (CFI)

M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti

Language-based Security
Dr. Kevin W. Hamlen
Motivation

• Goal: Implement In-lined Reference Monitors for x86 native code
  • Must prevent untrusted code from “jumping over” guard code
  • Must prevent untrusted code from overwriting guard code
  • Must prevent untrusted code from corrupting security state data

• Two policies to enforce:
  • Control-flow Integrity (constrain jumps)
  • Memory safety (constrain writes)

• Why are these two policies harder to enforce for IRMs in-lined into native code than they are for IRMs in-lined into Java bytecode?
Software Fault Isolation

• Enforce control-flow safety and memory safety

• Control-flow policy:
  • All reachable, in-module instructions appear in a static, fall-thru disassembly
  • Inter-module flows target exported function entrypoints
  • No jumps into middle of “chunks”

• Example Implementations:
  • PittSField [McCamant, Morrisett, USENIX Security ’06]
  • Google NaCl [Yee, Sehr, Dardyk, Chen, Muth, Ormandy, Okasaka, Narula, Fullagar, S&P ’09]
  • Reins [Wartell, Mohan, Hamlen, ACSAC ’12]
SFI Limitations

• Policy #1: The control-transfer instruction at address A may only call functions $f_1, f_2, \text{ or } f_3$
  • Why can’t traditional SFI enforce this policy?
  • What common source-level programming idiom gives rise to this policy?

• Policy #2: The return instruction at address A2 must target the instruction after the call that previously targeted A1
  • Why can’t traditional SFI enforce this policy? (at least two major reasons)
First Problem: Disassembly

- CISC native code consists of non-aligned, variable-length instruction encodings.
- Static disassembly is provably undecidable!
- Example: Disassemble these bytes: FF E0 5B 5D C3 0F 88 52 0F 84 EC 8B

<table>
<thead>
<tr>
<th>Valid Disassembly</th>
<th>Valid Disassembly</th>
<th>Valid Disassembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E0</td>
<td>jmp eax</td>
<td>FF E0</td>
</tr>
<tr>
<td>5B</td>
<td>pop ebx</td>
<td>5B</td>
</tr>
<tr>
<td>5D</td>
<td>pop ebp</td>
<td>5D</td>
</tr>
<tr>
<td>C3</td>
<td>retn</td>
<td>C3</td>
</tr>
<tr>
<td>0F 88 52</td>
<td>jcc</td>
<td>0F 88 52</td>
</tr>
<tr>
<td>0F 84 EC</td>
<td>mov</td>
<td>0F 84 EC</td>
</tr>
<tr>
<td>8B ...</td>
<td>mov</td>
<td>8B ...</td>
</tr>
</tbody>
</table>
Second Problem: Computed Jumps

• Many jump instructions compute their destinations at runtime – can potentially go *anywhere!*

• Examples:
  • jmp eax // start executing bytes at the address stored in eax
  • call eax // call a subroutine at address stored in eax
  • ret // load an address off the stack and jump to it

• IRM cannot safely impose guard code before dangerous operations if *any computed jump in the entire program* might jump over the guard code directly to the dangerous operation.
Third Problem: Writable Code, Executable Data

• By default, native code can write to any bytes in the address space – including its own code!
  • Cannot protect dangerous operations if any memory-write in the entire program might replace the guard code.

• By default, native code can jump to any bytes in the address space – including its data segment!
  • Cannot protect dangerous operations in runtime-generated code, since no guard code lives there.

• Hardware solution: Set code pages non-writable (NW) and data pages non-executable (NX)
  • How to prevent untrusted code from unsetting the protection bits?
CFI Workflow

**CFI:**

- Binary Code
- PDB File

**Policy:** CFG

**Rewriter**

**Safe Binary**

**Verify**
Control-Flow Integrity Policy

- **Static Control-Flow Graph (CFG)**
  - Derivable from application source code
  - Derivable from debug symbols (PDB file) yielded by Microsoft compilers
    - Avoids disclosure of full source code
    - Limits one to Microsoft-compiled code in practice
    - Requires code-producer cooperation!

- **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 );
}
```
Enforce the CFG

- Label jump targets with unique binary IDs
- Guard jumps with ID-checks

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF E1</td>
<td>jump 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>
</tbody>
</table>

can be instrumented as (a):

81 39 78 56 34 12 cmp [ecx], 12345678h ; comp ID & dst
75 13 jne error_label ; if != fail
8D 49 04 lea ecx, [ecx+4] ; skip ID at dst
FF E1 jump ecx ; jump to dst

or, alternatively, instrumented as (b):

B3 77 56 34 12 mov eax, 12345678h ; load ID-1
40 inc eax ; add 1 for ID
39 41 04 cmp [ecx+4], eax ; compare w/dst
75 13 jne error_label ; if != fail
FF E1 jump ecx ; jump to label
Requirements/Limitations

- Unique IDs
  - Must be able to find enough unique binary IDs not appearing in code
  - Not usually a problem in practice, but some tricky engineering problems

- Non-writable code
  - Use page-level write-protection
  - Runtime code self-modification not supported

- Non-executable data
  - Use Data Execution Prevention (DEP) NX-bit
  - Just-In-Time (JIT) compilation not supported (rules out many interpreters)
Limits of Static CFG Policies

• Call-return matching policy not expressible as CFG!

```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 );
}
```
Enforcing Call-Return Matching

• Enforce CFG to get uncircumventable guard code
• Use guard code to implement memory safety (SMAC)
• Use memory safety to implement a protected shadow-stack
  • Copy of the call stack that contains only the return addresses pushed by calls
  • Only protected guard code may write to it
• Reference shadow-stack to enforce call-return matching
Software Memory Access Control (SMAC)

• Goal: Write-protect certain memory regions from subsets of the code
  • Memory region is process-writable (e.g., so IRM can write to it)
  • But prohibit non-IRM code from writing to it (e.g., IRM integrity enforcement)

• Enforcement Strategy
  • Mask write-addresses
    • and eax, 0x0000FFFF
    • mov [eax], <data>
  • CFG-policy prevents circumvention of masking instruction

• Now we can implement secure data structures
  • Only writable by IRM
Call-return Matching

• Secure data structure: Shadow-stack
  • call L1
  • ...
  • L1: mov [shadow_stack], [esp]
  • inc shadow_stack_ptr

• Check shadow stack on returns
  • mov [esp], [shadow_stack]
  • dec shadow_stack_ptr
  • ret
Impact

• What happens if attacker exploits a buffer-overflow vulnerability to smash the stack?

• Caveat: Our experience is that most legacy Windows binaries do not obey call-return matching!
  • Tail-recursive calls
  • Exception-handling
  • Weird binary optimizations that don’t correspond to any source-level features
Microsoft’s Rewriting System

• Microsoft Vulcan
  • Multi-architecture rewriting
  • Requires .pdb file to accurately disassemble and analyze binary
Discussion

• What attacks continue to succeed against CFI?
• What attacks are thwarted?
• What are the challenges for widespread adoption?
• Compelling usage scenarios?