Control Flow Vulnerabilities:
ROP and CFI
Recall from last class

- Recall: Data Execution Prevention (DEP/W^X)
  - Prevent attacker input (which is data) from being interpreted as code by marking data pages as non-executable
    - One exception: applications like browsers that explicitly mark some of their data as executable to Just-In-Time compilation (JIT)
  - Win!

- Is there another way for an attacker to execute arbitrary code even without the ability to inject it into the victim process?
Code Reuse Attacks

- Use the code that’s already there

- What code is already there?
  - Program + shared libraries (including libc)
Return-To-Libc

▪ What can we find in libc?
  – “The system() library function uses fork(2) to create a child process that executes the shell command specified in command using execl(3) as follows:
    ```
    execl("/bin/sh", "sh", ":c", command, (char *) 0);
    ```
  – Need to find the address of:
    ▪ system()
    ▪ String “/bin/sh”
  – Overwrite the return address to point to start of system()
  – Place address of “/bin/sh” on the stack so that system() uses it as the argument
    ▪ To be clean, you also want to push exit() on the stack so it will shut down gracefully
Return-To-Libc

- What we want to get to
  - Transfer control to address of system() in libc
  - Setup stack frame to look like a normal call to system()
    - `int system(const char *command);`
  - `&exit()` system call is in the slot where the return address would be
  - `&cmd` is the argument
  - It points to the string “/bin/sh” stored further down the stack
Review: calling and returning

%ebp → main's locals
   3
   2
   1

%esp →

main()
-> foo(1,2,3)
   — — —> bar(4)
Review: calling and returning

%ebp → main’s locals
3
2
1

%esp → %eip in main

main() → foo(1,2,3)
   ─→ bar(4)
Review: calling and returning

%ebp →

<table>
<thead>
<tr>
<th>main’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>%eip in main</td>
</tr>
<tr>
<td>main’s %ebp</td>
</tr>
</tbody>
</table>

%esp →

main() -> foo(1,2,3)
        ----> bar(4)
Review: calling and returning

```
main() → foo(1,2,3) → bar(4)
```
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>%eip in main</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>main’s %ebp</td>
</tr>
<tr>
<td>2</td>
<td>foo’s locals</td>
</tr>
<tr>
<td>1</td>
<td>%ebp</td>
</tr>
<tr>
<td>%esp</td>
<td>%esp</td>
</tr>
</tbody>
</table>
Review: calling and returning

<table>
<thead>
<tr>
<th>%esp</th>
<th>%ebp</th>
<th>main’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>main’s %ebp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>foo’s locals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

main()
-> foo(1,2,3)
    — —> bar(4)
Review: calling and returning

main's locals
3
2
1
%eip in main
main's %ebp
foo's locals
4
%eip in foo

%ebp →
%esp →

main()
-> foo(1,2,3)
        ——> bar(4)
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>%ebp</th>
<th>esp</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
<td></td>
</tr>
<tr>
<td>main’s %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo’s locals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in foo</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo’s %ebp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

main() -> foo(1,2,3)  --> bar(4)
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>%eip in main</th>
<th>%ebp</th>
<th>main’s %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in foo</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%ebp → %esp

foo’s locals
| 4 | %eip in foo | foo’s %ebp |

main() -> foo(1,2,3)
        ——> bar(4)
Review: calling and returning

```
main's locals
  3
  2
  1
%eip in main
main's %ebp
foo's locals
  4
%eip in foo
foo's %ebp
bar's locals

%ebp ➔
%esp ➔

main() ➔ foo(1,2,3)
      ➔ bar(4)
```
Review: calling and returning

main() -> foo(1,2,3)  ----> bar(4)

leave = mov %ebp, %esp
       pop %ebp

%ebp %esp

main’s locals
3
2
1
%eip in main
main’s %ebp
foo’s locals
4
%eip in foo
foo’s %ebp
bar’s locals

%eip in main

%ebp

%esp
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>%esp → %ebp</th>
<th>leave = mov %ebp, %esp</th>
<th>pop %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>main’s %ebp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo’s locals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in foo</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo’s %ebp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar’s locals</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

main() -> foo(1,2,3) ▶ ▶ bar(4)
Review: calling and returning

```
main()

> foo(1,2,3)

---

> bar(4)

leave = mov %ebp, %esp
pop %ebp
```
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>%eip in main</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>foo’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
</tr>
<tr>
<td>%eip in foo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bar’s locals</th>
</tr>
</thead>
</table>

main() -> foo(1,2,3)  
             ——> bar(4)

ret = pop %eip
Review: calling and returning

<table>
<thead>
<tr>
<th>%ebp</th>
<th>%esp</th>
<th>main’s locals</th>
<th>foo’s locals</th>
<th>bar’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>%eip in main</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>main’s %ebp</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>foo’s locals</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>%eip in foo</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>foo’s %ebp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bar’s locals</td>
<td></td>
</tr>
</tbody>
</table>

main()  
-> foo(1,2,3)  
----> bar(4)
Review: calling and returning

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>main’s %ebp</th>
<th>foo’s locals</th>
<th>foo’s %ebp</th>
<th>bar’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

%esp → %ebp

leave = mov %ebp, %esp
pop %ebp

main()
    -> foo(1,2,3)
    --- > bar(4)
Review: calling and returning

<table>
<thead>
<tr>
<th>%ebp</th>
<th>main’s locals</th>
<th>%esp</th>
<th>main’s %ebp</th>
<th>foo’s locals</th>
<th>bar’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
<td>%eip in foo</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

main() -> foo(1,2,3)  
    -- -> bar(4)

leave =

mov %ebp, %esp
pop %ebp
## Review: calling and returning

<table>
<thead>
<tr>
<th>%ebp →</th>
<th>main’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>%eip in main</td>
</tr>
<tr>
<td></td>
<td>main’s %ebp</td>
</tr>
<tr>
<td></td>
<td>foo’s locals</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>%eip in foo</td>
</tr>
<tr>
<td></td>
<td>foo’s %ebp</td>
</tr>
<tr>
<td></td>
<td>bar’s locals</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>%esp →</th>
</tr>
</thead>
<tbody>
<tr>
<td>main()</td>
</tr>
<tr>
<td>→ foo(1,2,3)</td>
</tr>
<tr>
<td>→ bar(4)</td>
</tr>
</tbody>
</table>

**ret = pop %eip**
Review: calling and returning

main()
- > foo(1,2,3)
     --- > bar(4)
Suppose bar() had a stack overflow

- Our goal: call system("/bin/sh")
- Remember: Need to set up stack frame that looks like a legit call to system:

```
| cmd="/bin/sh" |
| &cmd |
| &exit |
```

- But we're not going to use the call instruction to jump to system; we're going to use ret
Hijacking control flow

<table>
<thead>
<tr>
<th>%ebp →</th>
<th>%esp →</th>
</tr>
</thead>
<tbody>
<tr>
<td>main’s locals</td>
<td>bar’s locals</td>
</tr>
<tr>
<td>3</td>
<td>%esp</td>
</tr>
<tr>
<td>2</td>
<td>%ebp</td>
</tr>
<tr>
<td>1</td>
<td>%eip in foo</td>
</tr>
<tr>
<td>main’s %ebp</td>
<td>foo’s %ebp</td>
</tr>
<tr>
<td>cmd=&quot;/bin/sh&quot;</td>
<td>&amp;system</td>
</tr>
<tr>
<td>%eip in foo</td>
<td>&amp;exit</td>
</tr>
<tr>
<td>foo’s locals</td>
<td>&amp;cmd</td>
</tr>
</tbody>
</table>

About to call leave
Hijacking control flow

- %eip in main
- main's %ebp
- foo's locals
- 4
- %eip in foo
- foo's %ebp
- bar's locals

<table>
<thead>
<tr>
<th>main's locals</th>
<th>cmd=&quot;/bin/sh&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

gets arguments from stack

Executing system()
Return To Libc

- Many different variants
- What else can attackers do by calling available functions with parameters of their choosing?
  - Move shellcode to unprotected memory
  - Change permissions on stack pages (`mprotect()`)
  - Etc.

Brendan Dolan-Gavitt
@moxy

Another CTF trick: if you need a string for `system()` that will get you a shell, consider the humble "ed". It supports running shell commands (!), and b/c of English past tense is often available as a suffix of some existing string in the binary, e.g.: "File transfer completed"
Return Oriented Programming (ROP)

- What if we cannot find just the right function? Or need more complex computation?
The Geometry of Innocent Flesh on the Bone:
Return-into-libc without Function Calls (on the x86)

Hovav Shacham*
hovav@cs.ucsd.edu
Return Oriented Programming

is a lot like a ransom note, but instead of cutting out letters from magazines, you are cutting out instructions from text segments.

ret Steve Checkoway
ret Dino Dai Zovi
Return Oriented Programming (ROP)

- What happens if we jump almost to the end of some function?
  - We execute the last few instructions, and then?
  - Then we return. But where?
  - To the return address on the stack. But we overwrote the stack with our own data so we control this address
    - Let’s choose to return to another tail of an existing function
    - Rinse and repeat
Return Oriented Programming (ROP)

- ROP idea: make complex shellcode out of existing application code

- Stitching together arbitrary programs out of code gadgets already present in the target binary
  - ROP Gadgets: code sequences ending in ret instruction.
  - Commonly added by compiler (at end of function)
  - But also (on x86) any sequence in executable memory ending in 0xC3 (ret).
    - x86 has variable-length instructions
    - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences
Aside: how Intel variable length instructions help ROP

- X86 instructions are variable length, yet can begin on any byte address

- Example:

```
81 c4 88 00 00 00 add $0x00000088, %esp
5f pop %edi
5d pop %ebp
c3 ret

00 5f 5d ad db addb %bl, 93 (%edi)
C3 ret
```

- Result: more “function tails” to choose from
Return Oriented Programming (ROP)

- ROP idea: make shellcode out of existing application code.

- Stitching together arbitrary programs out of code gadgets already present in the target binary
  - **ROP Gadgets**: code sequences ending in ret instruction.
  - Commonly added by compiler (at end of function)
  - But also (on x86) any sequence in executable memory ending in \(0xC3\) (ret).
    - x86 has variable-length instructions
    - Misalignment (jumping into the middle of a longer instruction) can produce new, unintended, code sequences

- Overwrite saved return address on stack to point to first gadget, the following word to point to second gadget, etc

- Stack pointer is the new instruction pointer in this crazy world
Return Oriented Programming

- “Our thesis: In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation.”
  - *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)* by Hovav Shacham
  - [https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf](https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf)

- Turing-complete computation.
  - Load and Store gadgets
  - Arithmetic and Logic gadgets
  - Control Flow gadgets
What does this gadget do?
relevant stack:

<table>
<thead>
<tr>
<th>%esp</th>
<th>\texttt{0x08049bbc}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{0xdeadbeef}</td>
<td>\texttt{0x08049bbc}</td>
</tr>
</tbody>
</table>

relevant register(s):

\texttt{%edx = 0x00000000}

relevant code:

\begin{verbatim}
%eip  \rightarrow  0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
0x08049bbd: ret
\end{verbatim}
relevant register(s):

%edx = 0x00000000

relevant stack:

%esp

- 0xdeadbeef
- 0x08049bbc

relevant code:

0x08049b62: nop
%eip → 0x08049b63: ret

... 0x08049bbc: pop %edx
0x08049b6d: ret
relevant stack:

%esp
0xdeadbeef
0x08049bbc

relevant register(s):
%edx = 0x00000000

relevant code:
0x08049b62: nop
0x08049b63: ret
...
%eip $0x08049bbc: pop %edx
0x08049bbd: ret
relevant register(s):
%edx = 0xdeadbeef

relevant stack:
%esp →
0xdeadbeef
0x08049bbc

relevant code:
0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
%eip → 0x08049b6d: ret
What does this gadget do?

%esp ➔ pop %edx ➔ ret

v₁ ➔ %edx = v₁

mov v₁, %edx
How do you use this as an attacker?

- Overflow the stack with values and addresses to such gadgets to express your program
- E.g., if shellcode needs to write a value to %edx, use the previous gadget
Return Oriented Programming

- Gadget for loading a value from memory
  - A bit more complex...
  - Attacker sets up stack so **address of value to be loaded** is on stack
    - Technically, 64 bytes less than addr
  - Return to gadget that pops that address into %eax
  - Return to gadget that loads the value based on address in %eax

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

- Control Flow Gadgets
  - Stack pointer is effectively the new instruction pointer
  - To “jump” just pop a new value into %esp
  - Conditional jumps are more involved but still possible

https://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
Return Oriented Programming

- Stack pointer acts as instruction pointer
- Manually stitching gadgets together gets tricky
  - Automation to the rescue!
  - Gadget finder tools, ROP chain compilers, etc.
  - All of this has been quasi-automated
    - UCSD built first ROP compiler as grad class project in 2009
    - As of 2013 CIA had a working ROP compiler toolchain (Vault7 leaks)

- Also: not even really about “returns”...
  other variants target other kinds of deterministic control flow (e.g. jump tables)

- Well, heck…. what to do?
Control Flow Integrity
Control Flow Integrity

- Given a new attack technique, we must present a new countermeasure.
  - Such is the cycle of life

- **Control Flow Integrity (CFI)**
  - Motivation: in almost all attacks we've seen attacker is overwriting jump targets (e.g., return address on stack, function pointers on heap, etc.)
    - What if we ensured that rets, calls, etc... could only go to *known good* targets
  - Basic idea: constraining the control-flow to only *legitimate paths determined in advance*
    - Match jump, call, and return sites to their target destinations
  - Many different implementations with different tradeoffs in protection strength and performance overhead.
Control Flow Integrity

- In almost all the attacks we looked at, the attacker is overwriting jump targets that are in memory (return addresses on the stack and function pointers on the stack/heap)

- **Idea:** don’t try to stop the memory writes. Instead: restrict control flow to legitimate paths
  - I.e., ensure that jumps, calls, and returns can only go to allowed target destinations
Control Flow Integrity

- Focus is on protecting indirect transfer of control flow instructions.

- *Direct* control flow transfer:
  - Advancing to next sequential instruction
  - Jumping to (or calling a function at) an address hard-coded in the instruction
  - These are static in code, so assume attacker can’t control (if they can overwrite code segment its game over anyway)

- *Indirect* control flow transfer
  - Jumping to (or calling a function at) an address in register or memory
  - *Forward path*: indirect calls and branches (e.g., a function you are calling)
  - *Reverse path*: return addresses on the stack (returning from a called function)
What’s a legitimate target?

Look at the program control-flow graph (CFG)!

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

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

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

```c
sort2()

sort()

call sort

lt()

call arg$3

ret

gt()

ret

direct call

indirect call

return
```
Control Flow Integrity

▪ Basic Design:
  – Restrict all control transfers to the control flow graph
  – Assign **labels** to all indirect jumps and their targets
  – Before taking an indirect jump, validate that target label matches jump site
    ▪ Like stack canaries, but for control flow targets
  – Can use hardware support is essential to make enforcement efficient
    ▪ Absent that, make performance/precision tradeoffs
**CFI: Example of Labels**

**Original code**

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

**Instrumented code**

- Jump to the destination only if the tag is equal to “12345678”
- Abuse an x86 assembly instruction to insert “12345678” tag into the binary
Fine grained CFI (Abadi et al.)

- Statically compute CFG

- Dynamically ensure program never deviates
  - Assign label to each target of indirect control transfer
  - Instrument indirect control transfers to compare label of destination with the expected label to ensure it's valid
Fine grained CFI (Abadi et al.)

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

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

bool gt(int x, int y) {
    return x > y;
}
Fine grained CFI (Abadi et al.)

- Statically compute CFG
- Dynamically ensure program never deviates
  - Assign label to each target of indirect transfer
  - Instrument indirect transfers to compare label of destination with the expected label to ensure it’s valid
- Note, returns are aliased in this example
  - Can return to multiple places, even if not “correct”
- Can do a bit better with a “shadow stack” (part of original CFI proposal)
  - Second, protected stack just for control data, to ensure that you are returning to function you called from
Shadow Stack

- Shadow Stack
  - On function entry, save a [shadow] copy of function call control flow data (return addresses and frame pointers) into another location
  - On function exit, compare the version on the stack to the shadow copy

- Requires compiler support

- Requires hardware support to be fast
  - Intel CET Shadow Stack starting with Tiger Lake

- Not widely deployed/available
Coarse-grained CFI (e.g., bin-CFI)

- Tradeoff speed for precision
  - Identify if control transfer is *clearly wrong*, but not that it is right
    - i.e., can only transfer control to a legitimate destination in the CFG, but don’t check which one
  - Only two labels, no shadow stack

- Label for destination of indirect calls (forward)
  - Make sure that every indirect call lands on a function entry

- Label for destination of rets and indirect jumps (reverse)
  - Make sure every indirect jump lands at start of a basic block
CFI Tradeoffs and Bypass Opportunities

- **Overhead**
  - Additional computation is needed before every free branch instruction.
  - Additional code size is needed before every free branch instruction and at each location (the label).

- **Scope**
  - Data is not protected
  - CFI does not protect against interpreters
  - Needs reliable DEP (if you can change code all bets are off)

- **Precision**
  - If you don’t validate all data-dependent control transfers, can still create gadgets
    - E.g., can still call system() with coarse grained CFI
    - Lots of potential gadgets on return path (you really need a Shadow Stack)
  - Performance/Precise tradeoff creates holes
    - Out of Control: Overcoming Control-Flow Integrity
    - Stitching the Gadgets: On the Ineffectiveness of Coarse Grained CFI
  - Aside: C++ is a huge huge problem due to virtual functions

- Some version of CFI is used today on both Apple iOS and Google Android
Summary

- Code reuse attacks bypass DEP by using existing code
  - E.g., Return to libc

- Return-oriented Programming
  - Generalizes idea. You can synthesize arbitrary malicious computation out benign code, by stitching it together with control flow
  - The stack is a convenient place to do this stitching

- Control-flow integrity
  - Promise to provide general-purpose protection against control flow vulnerabilities
  - Gets part of the way there at significant cost, but still bypassable due to limited precision
Next Lecture...

System Security