CSE 127: Computer Security

ROP, heap attacks, CFI, integer overflows

Nadia Heninger and Deian Stefan

Some slides adopted from Kirill Levchenko, Stefan Savage, Stephen Checkoway, Hovav Shacham, Raluca Popal, and David Wagner
Review: calling and returning

%ebp → main’s locals
%esp →

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

%ebp → main’s locals

%esp →

3

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

%ebp → main’s locals

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></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>

%esp →

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

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

--> bar(4)
Review: calling and returning

main

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

---> bar(4)
Review: calling and returning

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

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

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

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

%esp points to the %ebp in main.
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>%eip in foo</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>%ebp</th>
<th>main’s %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>main’s %ebp</td>
<td>foo’s %ebp</td>
</tr>
<tr>
<td>foo’s locals</td>
<td>foo’s %ebp</td>
</tr>
</tbody>
</table>

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

Call stack:
- main
- foo
- bar
Review: calling and returning

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

`main()`

-> `foo(1,2,3)`

--> `bar(4)`
Review: calling and returning

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

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

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

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

leave = mov %ebp, %esp
       pop %ebp

%esp → %ebp →

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

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

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


Review: calling and returning

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

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

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

--> bar(4)

ret = pop %eip
Review: calling and returning

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

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

leave = mov %ebp, %esp
pop %ebp

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>%esp → %ebp</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>
<tr>
<td>%eip in main</td>
<td></td>
</tr>
<tr>
<td>main’s %ebp</td>
<td></td>
</tr>
<tr>
<td>foo’s locals</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>%eip in foo</td>
<td></td>
</tr>
<tr>
<td>foo’s %ebp</td>
<td></td>
</tr>
<tr>
<td>bar’s locals</td>
<td></td>
</tr>
</tbody>
</table>
Review: calling and returning

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

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

ret = pop %eip
Review: calling and returning

\[ \text{%ebp} \rightarrow \text{main's locals} \]

\[
\begin{array}{c}
3 \\
2 \\
1 \\
\text{%eip in main} \\
\text{main's %ebp} \\
\text{foo's locals} \\
4 \\
\text{%eip in foo} \\
\text{foo's %ebp} \\
\text{bar's locals}
\end{array}
\]

main()
\[ \rightarrow \text{foo(1,2,3)} \]

\[ \rightarrow \text{bar(4)} \]
Suppose bar had overflow

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

```
| cmd="/bin/sh" |
| &cmd          |
| saved %eip    |
```

- But we're not going to use call instruction to jump to system; we're going to use ret
Suppose bar had overflow

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

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

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

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>foo’s locals</th>
<th>bar’s locals</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
<td>%esp</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>%ebp</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>%esp</td>
</tr>
<tr>
<td>%eip in main</td>
<td>%eip in foo</td>
<td>%ebp</td>
</tr>
<tr>
<td>main’s %ebp</td>
<td>foo’s %ebp</td>
<td>%ebp</td>
</tr>
<tr>
<td>cmd=/bin/sh</td>
<td>&amp;cmd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&amp;exit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp;system</td>
</tr>
</tbody>
</table>
Hijacking control flow

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

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

%ebp →

%esp →

bar’s locals

ret
## Hijacking control flow

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

- cmd="/bin/sh"
- &cmd
- &exit
- &system
Hijacking control flow

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

<table>
<thead>
<tr>
<th>main’s locals</th>
<th>cmd=&quot;/bin/sh&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>&amp;cmd</td>
</tr>
<tr>
<td>2</td>
<td>&amp;exit</td>
</tr>
<tr>
<td>1</td>
<td>&amp;system</td>
</tr>
<tr>
<td>%eip in main</td>
<td></td>
</tr>
<tr>
<td>main’s %ebp</td>
<td></td>
</tr>
<tr>
<td>foo’s locals</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>%eip in foo</td>
<td></td>
</tr>
<tr>
<td>foo’s %ebp</td>
<td></td>
</tr>
<tr>
<td>bar’s locals</td>
<td></td>
</tr>
</tbody>
</table>

points to nonsense, but doesn't matter; system just saves it
Hijacking control flow

- Stack frame that looks like a normal call to `system`:

```
%ebp
%esp
```

```c
cmd="/bin/sh"
&cmd
&exit
```
Today

• Advanced modern attack techniques
  ➢ ROP
  ➢ Heap-based attacks
• Control flow integrity
• Integer overflow attacks
Employees must wash hands before returning to libc
What if there is no code that does what we want?
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.
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
Return-Oriented Programming

- Idea: make shellcode out of existing code
- Gadgets: code sequences ending in ret instruction
  - Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

• Where do you often find ret instructions?
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

• Where do you often find ret instructions?
  ➤ End of function (inserted by compiler)
Return-Oriented Programming

• Idea: make shellcode out of existing code

• Gadgets: code sequences ending in ret instruction
  ➤ Overwrite saved %eip on stack to pointer to first gadget, then second gadget, etc.

• Where do you often find ret instructions?
  ➤ End of function (inserted by compiler)
  ➤ Any sequence of executable memory ending in 0xc3
x86 instructions

- Variable length!
- Can begin on any byte boundary!
One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3

=  
mov $0x1,%eax
pop %ebx
leave
ret
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3
```

= add %al,(%eax)
pop %ebx
leave
ret
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3 = add %bl, -0x37(%eax)
ret
```
One ret, multiple gadgets

```
b8 01 00 00 00 5b c9 c3  =  pop %ebx
       ____  
leave
ret
```
One ret, multiple gadgets

\[
\text{b8 01 00 00 00 5b } \quad \underline{c9 \quad c3} \quad = \quad \text{leave} \quad \text{ret}
\]
One ret, multiple gadgets

b8 01 00 00 00 5b c9 c3  =  ret
What does this gadget do?

%esp → v₁ → pop %edx → ret
relevant stack:

%esp → 0xdeadbeef
    0x08049bbc

relevant register(s):

%edx = 0x00000000

relevant code:

%eip → 0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
0x08049bbd: ret
relevant stack:

<table>
<thead>
<tr>
<th>%esp</th>
<th>0x08049bbc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xdeadbeef</td>
<td></td>
</tr>
</tbody>
</table>

relevant register(s):

%edx = 0x00000000

relevant code:

0x08049b62: nop

%eip → 0x08049b63: ret

... 

0x08049bbc: pop %edx
0x08049bbd: ret
relevant stack:

%esp →

| 0xdeadbeef |
| 0x08049bbc |

relevant code:

0x08049b62: nop
0x08049b63: ret
...

%eip →

0x08049bbc: pop %edx
0x08049bbd: ret

relevant register(s):

%edx = 0x00000000
relevant register(s):

%edx = 0xdeadbeef

relevant stack:

%esp →
0xdeadbeef
0x08049bbc

relevant code:

0x08049b62: nop
0x08049b63: ret
...
0x08049bbc: pop %edx
%eip → 0x08049bbd: ret
What does this gadget do?

%esp

\[ v_1 \]

\[ \text{pop } \%edx \]

\[ \text{ret} \]

\[ %edx = v_1 \]

\[ \text{mov } v_1, \%edx \]
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
What does this gadget do?

%esp → pop %eax
    → ret
v1 → pop %ebx
    → ret
v2 → mov %eax, (%ebx)
    → ret
relevant register(s):

- %eax = 0x00000000
- %ebx = 0x00000000

relevant stack:

- %esp
- 0x08049b90
- 0xbadcaffe
- 0x08049b63
- 0xdeadbeef
- 0x08049bbc

relevant memory:

- 0xbadcaffe: 0x00000000

relevant code:

- %eip
- 0x08049b00: ret
  ...
- 0x08049b63: pop %ebx
- 0x08049b64: ret
  ...
- 0x08049b90: mov %eax, %ebx
- 0x08049b91: ret
  ...
- 0x08049bbc: pop %eax
- 0x08049bbd: ret
relevant register(s):

\[
\begin{align*}
%eax &= 0x00000000 \\
%ebx &= 0x00000000
\end{align*}
\]

relevant memory:

\[
0xbadcaffe: \ 0x00000000
\]

relevant code:

\[
\begin{align*}
0x08049b90: \ & \text{mov} \ %eax, \ %(%ebx) \\
0x08049b91: \ & \text{ret} \\
0x08049b9b: \ & \text{ret} \\
0x08049b9d: \ & \text{pop} \ %ebx \\
0x08049b63: \ & \text{pop} \ %ebx \\
0x08049b64: \ & \text{ret} \\
0x08049b66: \ & \text{ret} \\
0x08049b68: \ & \text{ret} \\
0x08049b6a: \ & \text{ret} \\
0x08049b6c: \ & \text{ret} \\
0x08049b6e: \ & \text{ret} \\
0x08049b70: \ & \text{ret} \\
0x08049b72: \ & \text{ret} \\
0x08049b74: \ & \text{ret} \\
0x08049b76: \ & \text{ret} \\
0x08049b78: \ & \text{ret} \\
0x08049b7a: \ & \text{ret} \\
0x08049b7c: \ & \text{ret} \\
0x08049b7e: \ & \text{ret} \\
0x08049b80: \ & \text{ret} \\
0x08049b82: \ & \text{ret} \\
0x08049b84: \ & \text{ret} \\
0x08049b86: \ & \text{ret} \\
0x08049b88: \ & \text{ret} \\
0x08049b8a: \ & \text{ret} \\
0x08049b8c: \ & \text{ret} \\
0x08049b8e: \ & \text{ret} \\
0x08049b90: \ & \text{ret} \\
0x08049b92: \ & \text{ret} \\
0x08049b94: \ & \text{ret} \\
0x08049b96: \ & \text{ret} \\
0x08049b98: \ & \text{ret} \\
0x08049b9a: \ & \text{ret} \\
0x08049b9c: \ & \text{ret} \\
0x08049b9e: \ & \text{ret} \\
0x08049b00: \ & \text{ret} \\
\end{align*}
\]

%eip: 0x08049bbc: pop %eax
\[
0x08049b63: \ & \text{pop} \ %ebx \\
0x08049b64: \ & \text{ret} \\
0x08049b66: \ & \text{ret} \\
0x08049b68: \ & \text{ret} \\
0x08049b6a: \ & \text{ret} \\
0x08049b6c: \ & \text{ret} \\
0x08049b6e: \ & \text{ret} \\
0x08049b80: \ & \text{ret} \\
0x08049b82: \ & \text{ret} \\
0x08049b84: \ & \text{ret} \\
0x08049b86: \ & \text{ret} \\
0x08049b88: \ & \text{ret} \\
0x08049b8a: \ & \text{ret} \\
0x08049b8c: \ & \text{ret} \\
0x08049b8e: \ & \text{ret} \\
0x08049b90: \ & \text{ret} \\
0x08049b92: \ & \text{ret} \\
0x08049b94: \ & \text{ret} \\
0x08049b96: \ & \text{ret} \\
0x08049b98: \ & \text{ret} \\
0x08049b9a: \ & \text{ret} \\
0x08049b9c: \ & \text{ret} \\
0x08049b9e: \ & \text{ret} \\
0x08049b00: \ & \text{ret} \\
\]
relevant register(s):

%eax = 0xdeadbeef
%ebx = 0x00000000

relevant stack:

%esp →

| 0x08049b90    |
| 0xbadcaffe    |
| 0x08049b63    |
| 0xdeadbeef    |
| 0x08049bbc    |

relevant memory:

0xbadcaffe: 0x00000000

relevant code:

0x08049b00: ret
... 0x08049b63: pop %ebx 0x08049b64: ret...
... 0x08049b90: mov %eax, %([ebx]) 0x08049b91: ret...
... 0x08049bbc: pop %eax %eip → 0x08049bbd: ret
### Relevant Register(s):

- \( \%eax = 0x\text{deadbeef} \)
- \( \%ebx = 0x00000000 \)

### Relevant Memory:

- 0xbadcaffe: 0x00000000

### Relevant Code:

```assembly
0x08049b00: ret
...  
0x08049b63: pop \%ebx
0x08049b64: ret
...  
0x08049b90: mov \%eax, \%(ebx)
0x08049b91: ret
...  
0x08049bbc: pop \%eax
0x08049bbd: ret
```

### Relevant Stack:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%esp</td>
<td>0x08049b90</td>
</tr>
<tr>
<td>0xbadcaffe</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0xdeadbeef</td>
<td></td>
</tr>
<tr>
<td>0x08049bbc</td>
<td></td>
</tr>
</tbody>
</table>
relevant register(s):
%eax = 0xdeadbeef
%ebx = 0xbadcaffe

relevant stack:
%esp 0x08049b90 0xbadcaffe 0x08049b63 0xdeadbeef 0x08049bbc

relevant memory:
0xbadcaffe: 0x00000000

relevant code:
0x08049b00: ret
... 0x08049b63: pop %ebx
0x08049b64: ret
... 0x08049b90: mov %eax, %(%ebx) 0x08049b91: ret 0x08049bbc: pop %eax 0x08049bbd: ret
relevant register(s):

%eax = 0xdeadbeef
%ebx = 0xbadcaffe

relevant stack:

%esp

0x08049b90
0xbadcaffe
0x08049b63
0xdeadbeef
0x08049bbc

relevant memory:

0xbadcaffe: 0x00000000

relevant code:

0x08049b00: ret
...
0x08049b63: pop %ebx
0x08049b64: ret
...
%eip

0x08049b90: mov %eax, %(%ebx)
0x08049b91: ret
...
0x08049bbc: pop %eax
0x08049bbd: ret
relevant register(s):

- %eax = 0xdeadbeef
- %ebx = 0xbadcaffe

relevant stack:

- %esp
- 0x08049b90
- 0xbadcaffe
- 0x08049b63
- 0xdeadbeef
- 0x08049bbc

relevant memory:

- 0xbadcaffe: 0xdeadbeef

relevant code:

- 0x08049b00: ret
- 0x08049b63: pop %ebx
- 0x08049b64: ret
- 0x08049b90: mov %eax, %([ebx])
- 0x08049b91: ret
- 0x08049bbc: pop %eax
- 0x08049bbd: ret
What does this gadget do?

\[
\begin{align*}
\text{mem}[v_2] &= v_1 \\
\text{mov v}_2, \%ebx \\
\text{mov v}_1, \%(%ebx)
\end{align*}
\]
Can express arbitrary programs

Figure 5: Simple add into %eax.

Figure 16: Shellcode.

Figure 10: An infinite loop by means of an unconditional jump.
Can find gadgets automatically

Hacking Blind

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, Dan Boneh
Stanford University

Ropper - rop gadget finder and binary information tool

You can use ropper to look at information about files in different file formats and you can find ROP and JOP gadgets to build chains for different architectures. Ropper supports ELF, MachO and the PE file format. Other files can be opened in RAW format. The following architectures are supported:

- x86 / x86_64
- Mips / Mips64
- ARM (also Thumb Mode) / ARM64
- PowerPC / PowerPC64
Return-Oriented Programming

not even really about “returns”...
Today

• Advanced modern attack techniques
  ➢ ROP
  ➢ Heap-based attacks
• Control flow integrity
• Integer overflow attacks
Handling heap-allocated memory can be just as error-prone as the stack

• We may:
  ➤ Write/read memory we shouldn’t have access to
  ➤ Forget to free memory
  ➤ Free already freed objects
  ➤ Use pointers that point to freed object

• What if the attacker can cause the program to use freed objects?
Heap corruption

• Can bypass security checks (data-only attacks)
  ➤ E.g., isAuthenticatted, buffer_size, isAdmin, etc.

• Can overwrite function pointers
  ➤ Direct transfer of control when function is called
  ➤ C++ virtual tables are especially good targets
vtables

• Each object contains pointer to vtable

• Array of function pointers
  ➢ one entry per function

• Call looks up entry in vtable

Q: What does `bar()` compile to?

A: `*(obj->vtable[0])(obj)`

```cpp
class Base {
    public: virtual void foo() {
        cout << "Hi\n";
    }
};

class Derived: public Base {
    public: void foo() {cout << "Bye\n";}
};

void bar(Base* obj) { obj->foo(); }

int main(int argc, char* argv[]) {
    Base *b = new Base();
    Derived *d = new Derived();
    bar(b);
    bar(d);
}
```
What does a use after free (UAF) attack look like?

**Victim:** Free object: `free(obj);`

**Attacker:** Overwrite the vtable of the object so entry (e.g., `obj->vtable[0]`) points to attacker gadget

**Victim:** Use dangling pointer: `obj->foo()`
Today

• **Advanced modern attack techniques**
  ➢ ROP
  ➢ Heap-based attacks

• **Control flow integrity**

• **Integer overflow attacks**
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
Restrict indirect transfers of control
Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?
Restrict indirect transfers of control

- Why do we not need to do anything about direct transfer of control flow (i.e., direct jumps/calls)?
  - Address is hard-coded in instruction. Not under attacker control
Restrict indirect transfers of control
Restrict indirect transfers of control

• What are the ways to transfer control indirectly?
Restrict indirect transfers of control

• What are the ways to transfer control indirectly?

• **Forward path**: jumping to (or calling function at) an address in register or memory
  ➤ E.g., qsort, interrupt handlers, virtual calls, etc.

• **Reverse path**: returning from function (uses address on stack)
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;
}
```
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;
}
```
What’s a legitimate target?

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

```c
typedef int Array[200];

void sort2(Array a, Array 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;
}
```
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;
}
```
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;
}
```

<table>
<thead>
<tr>
<th>sort2()</th>
<th>sort()</th>
<th>lt()</th>
<th>gt()</th>
</tr>
</thead>
<tbody>
<tr>
<td>call sort</td>
<td>call arg$3</td>
<td>ret</td>
<td>ret</td>
</tr>
</tbody>
</table>

- `-` -> direct call
- `-----` -> indirect call
- `←` ← return
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;
}
```

```
            sort2()  |  sort()  |   lt()   |   gt()   |
            +----------+----------+----------+----------|
            | call sort | call arg$3| ret | ret |
            +----------+----------+----------+----------|
```

- direct call
- indirect call
- return
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;
}
```
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;
}
```

![Control Flow Graph](image)
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;
}
```
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;
}
```
How do we restrict jumps to CFG?

• 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 target
• Need hardware support
  ➤ Otherwise trade off precision for performance
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
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;
}
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;
}
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.)

```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;
}
```
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;
}
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.)

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;
}
Coarse-grained CFI (bin-CFI)

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

- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of BB
Coarse-grained CFI (bin-CFI)

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

- Label for destination of rets and indirect jumps
  - Make sure every indirect jump lands at start of BB
How else can you choose labels?

\[
\frac{tf = t_1^* \rightarrow t_2^* \quad C_{\text{table}} = n}{C \vdash \text{call\_indirect } tf : t_1^* \rightarrow t_2^*}
\]

\[
s; (i32.\text{const } j) \text{ call\_indirect } tf \quad \leftarrow_i \quad \text{call } s_{\text{tab}}(i, j)
\]

\[
s; (i32.\text{const } j) \text{ call\_indirect } tf \quad \leftarrow_i \quad \text{trap}
\]

\[
\text{if } s_{\text{tab}}(i, j)_{\text{code}} = (\text{func } tf \text{ local } t^* \text{ e}^*) \quad \text{otherwise}
\]
How else can you choose labels?

WebAssembly does it by looking at function type

\[
\frac{tf = t_1^* \rightarrow t_2^* \quad C_{table} = n}{C \vdash \text{call\_indirect } tf : t_1^* \text{i32} \rightarrow t_2^*}
\]

\[
s; (i32.\text{const } j) \text{call\_indirect } tf \quad \mapsto_i \quad \text{call } s_{\text{tab}}(i, j)
\]

\[
s; (i32.\text{const } j) \text{call\_indirect } tf \quad \mapsto_i \quad \text{trap}
\]

if \(s_{\text{tab}}(i, j)_{\text{code}} = (\text{func } tf \text{ local } t^* e^*)\)
otherwise
What do labels look like?

**Original code**

<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><code>jmp ebx</code> ; computed jump</td>
<td>8B 44 24 04</td>
<td><code>mov eax, [esp+4]</code> ; dst</td>
</tr>
</tbody>
</table>
What do labels look like?

### 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>mov eax, [esp+4]; dst</td>
</tr>
</tbody>
</table>

### Instrumented code

```
48 77 56 34 12  mov  eax, 12345677h ; 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  jmp  ecx       ; jump to label
```
What do labels look like?

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

**Original code**

**Instrumented code**

Abuse an x86 assembly instruction to insert “12345678” tag into the binary.
What do labels look like?

**Original code**

<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>jmp ecx</td>
<td>8B 44 24 04</td>
<td>mov eax, [esp+4]</td>
</tr>
</tbody>
</table>

; computed jump

**Instrumented code**

<table>
<thead>
<tr>
<th>Opcode bytes</th>
<th>Source Instructions</th>
<th>Opcode bytes</th>
<th>Destination Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>B8 77 56 34 12</td>
<td>inc eax, 12345677h</td>
<td>3E 0F 18 05</td>
<td>prefetchnta</td>
</tr>
<tr>
<td>40</td>
<td>inc eax</td>
<td>73 56 34 12</td>
<td>[12345677h]</td>
</tr>
<tr>
<td>39 41 04</td>
<td>cmp [ecx+4], eax</td>
<td>3B 44 24 04</td>
<td>mov eax, [esp+4]</td>
</tr>
<tr>
<td>75 13</td>
<td>jne error_label</td>
<td>...</td>
<td>dst</td>
</tr>
<tr>
<td>FF E1</td>
<td>jmp ecx</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump to the destination only if the tag is equal to “12345678”

Abuse an x86 assembly instruction to insert “12345678” tag into the binary
CFI limitations

• Overhead
  ➤ Runtime: every indirect branch instruction
  ➤ Size: code before indirect branch + encode label at destination

• Scope
  ➤ CFI does not protect against data-only attacks
  ➤ Needs reliable W^X
How can you defeat CFI?

- Imprecision can allow for control-flow hijacking
  - Can jump to functions that have same label
    - E.g., even if we use Wasm’s labels `int system(char*)` and `int myFunc(char*)` share the same label
  - Can return to many more sites
    - But, real way to do backward edge CFI is to use a shadow stack. (This is actually great!)
Today

• Advanced modern attack techniques
  ➤ ROP
  ➤ Heap-based attacks

• Control flow integrity

• Integer overflow attacks
What’s wrong with this program?

void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
What’s wrong with this program?

```c
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

MEMCPY(3)  Linux Programmer’s Manual  MEMCPY(3)

NAME  top

memcpy - copy memory area

SYNOPSIS top

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```
What’s wrong with this program?

```c
void vulnerable(int len, char *data) {
    char buf[64];
    if (len > 64)
        return;
    memcpy(buf, data, len);
}
```

### MEMCPY(3) Linux Programmer’s Manual

#### NAME

`memcpy` - copy memory area

#### SYNOPSIS

```c
#include <string.h>

void *memcpy(void *dest, const void *src, size_t n);
```
What’s wrong with this program?

```c
void vulnerable(int len = 0xffffffff, char *data) {
    char buf[64];
    if (len = -1 > 64)
        return;
    memcpy(buf, data, len = 0xffffffff);
}
```

---

**NAME**

`memcpy` - copy memory area

**SYNOPSIS**

```c
#include <string.h>
void *memcpy(void *dest, const void *src, size_t n);
```
Is this program safe?

```c
void f(size_t len, char *data) {
    char *buf = malloc(len+2);
    if (buf == NULL)
        return;
    memcpy(buf, data, len);
    buf[len] = '\n';
    buf[len+1] = '\0';
}
```
Is this program safe?

No!

```c
void f(size_t len = 0xffffffff, char *data) {
    char *buf = malloc(len+2 = 0x000000001);
    if (buf == NULL)
        return;
    memcpy(buf, data, len = 0xffffffff);
    buf[len] = ‘\n’;
    buf[len+1] = ‘\0’;
}
```
Still relevant classes of bugs

Issue 952406: Security: Possible OOB related to chrome_sqlite3_malloc
Reported by mlfbr_@stanford.edu on Fri, Apr 12, 2019, 1:59 PM PDT

VULNERABILITY DETAILS
Possible OOB with chrome_sqlite3_malloc

REPRODUCTION CASE
There’s a pattern of using sqlite malloc functions that call chrome_sqlite3_malloc in combination with traditional memory operations (e.g., memcpy). There may be invariants that make this ok, or a principle here that I am not aware of. Thanks for your time.

**chrome_sqlite3_malloc** takes an **int size** argument, while **memcpy** takes a **size_t size** argument. On x86-64 this means that chrome_sqlite3_malloc's size argument is width 32, while memcpy's is width 64. This can lead to potentially concerning wrapping behavior for extreme allocation sizes (depending on the compiler, optimizations, etc).

For example:

Function fts3UpdateDocTotals

(1) a = sqlite3_malloc( (sizeof(u32)+10)*nStat );
...
(2) memset(a, 0, sizeof(u32)*nStat);

Depending on optimization level etc, this may turn into:

(1)
size = mul i32 nstat 14
chrome_sqlite3_malloc(size)
Three flavors of integer overflows

- **Truncation bugs**
  - E.g., assigning an int64_t into in32_t (3rd ex)

- **Arithmetic overflow bugs**
  - E.g., adding huge unsigned number (2nd ex)

- **Signedness bugs**
  - E.g., treating signed number as unsigned (1st ex)
Today

- Advanced modern attack techniques
  - ROP
  - Heap-based attacks
- Control flow integrity
- Integer overflow attacks
What does this all tell us?

If you’re trying to build secure systems, use a memory safe language.