CMPSC 447
Return-oriented Programming

Trent Jaeger
Systems and Internet Infrastructure Security (SIIS) Lab
Computer Science and Engineering Department
Pennsylvania State University
Code Injection

- Remember this exploit
- The adversary’s goal is to get `execve` to run to generate a command shell
- To do this the adversary uses `execve` from libc – i.e., reuses code that is already there

```
stack frame for main

execve ("/bin/sh")
ret
1
2
```

Injection Requirements

- What is **required** for a code injection attack?
  - Appreciated by the adversary…
  - That is **not expected** in practice?

---

*Gratuity*

**APPRECIATED**

**BUT NOT EXPECTED**
Injection Requirements

• What is required for a code injection attack?
  ‣ Appreciated by the adversary…
  ‣ That is not expected in practice?

• **Answer**: Execute stack memory
  ‣ Code is injected in stack memory
  ‣ So, we must be able to execute stack memory

• Must all memory be executable?
  ‣ Recall page permissions
Prevent Injection

• An available defense can prevent injection
  ‣ DEP or W xor X: Stack memory is not executable

• Set the program memory regions to be either writable or executable, but not both
  ‣Writable: Stack and heap and global data
  ‣ Executable: Code
  ‣ Of course, some can be read-only and not executable

• Bottom line is that we can remove the execute permission from stack and heap memory pages
  ‣ And prevent writing of code pages
Bypass DEP

- Can we *invoke execve without code injection*?
  - If so, how?
Return-to-execve

- How can we *invoke execve without code injection*?
  - Use the code directly
- The difference is subtle, but significant

![Diagram showing stack frames and function calls involving execve and plt. The diagram illustrates how the return address is altered to execute a different program at the same memory location.](image-url)
Return-to-execve

• How can we invoke execve without code injection?
  ‣ Call execve directly from return value

• The difference is subtle, but significant
  ‣ In the original exploit, we wrote the address of execve into buffer on the stack and modified return address to start executing at buffer
  • I.e., we are executing in the stack memory region
  ‣ Instead, we can modify the return address to point to execve directly, so we continue to execute code
  • Key: Point return address (function pointer) to code memory (PLT to invoke libc function) rather than stack memory
Return-to-Libc

• Can we invoke any Libc function without code injection?
  ‣ Well, any that the program uses explicitly from the PLT
  ‣ And any other from Libc code – if you know where it is

• Called “Return-to-Libc” in general
  ‣ Change the return address to refer to a Libc function
  ‣ Gives you access to a lot of valuable code for attacks

• Can you invoke other code like this?
Return-to-X in General

- Return-to-Libc attacks can be employed more generally to enable adversaries to execute existing code under their control

  - Termed “return-oriented attacks”
    - by Hovav Shacham and his colleagues
    - Next few slides are Prof Shacham’s
Bad code versus bad behavior

“Bad” behavior

Attacker code

Problem: this implication is false!

“Good” behavior

Application code
any sufficiently large program codebase

arbitrary attacker computation and behavior, 
*without* code injection

(in the absence of control-flow integrity)
Code Reuse in General
• Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  ‣ Termed "return-oriented attacks"

ROP vs return-to-libc

attacker control of stack

arbitrary attacker computation and behavior via return-into-libc techniques

(given any sufficiently large codebase to draw on)
Machine Instructions

- Instruction pointer (%eip) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments %eip to next instruction
- Control flow by changing value of %eip
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed “return-oriented attacks”

ROP Execution

- **Stack pointer** (%esp) determines which instruction sequence to fetch & execute
- Processor doesn’t automatically increment %esp; — but the “ret” at end of each instruction sequence does

`insns ... ret`

`C library`

`stack pointer`
**Code Reuse in General**

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks" (ROP)

**Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1: pop %eax
    ret
G2: pop %ebx
    ret
G3: movl %eax, (%ebx)
    ret
```

**Diagram**

- **Registers**
  - %eax = 
  - %ebx =

- **Stack**
  - G1
  - 5
  - G2
  - 0x8048000
  - G3
  - ...

- **Memory**
  - 0x8048000 = buf
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks"

**Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:  pop %eax
    ret

G2:  pop %ebx
    ret

G3:  movl %eax, (%ebx)
    ret
```

**Registers**

- %eax = 5
- %ebx =

**Memory**

- 0x8048000 =

**Stack**

- G1
- 5
- G2
- 0x8048000
- G3
- ...

**Return Address**

- buf

**Diagram**

- Code block highlighting G1 with instructions: pop %eax, ret
- Stack diagram showing return address and values:
  - G1
  - 5
  - G2
  - 0x8048000
  - G3
  - ...
  - buf
- Memory block showing 0x8048000 value.
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks" (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:      pop %eax
        ret

G2:      pop %ebx
        ret

G3:      movl %eax, (%ebx)
        ret
```

![Diagram showing the process of code reuse with ESP as program counter and the stack and memory setup.](image)
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed “return-oriented attacks” (ROP)

Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```c
G1: pop %eax
    ret

G2: pop %ebx
    ret

G3: movl %eax, (%ebx)
    ret
```

**Registers**

- `%eax = 5`
- `%ebx = 0x8048000`

**Memory**

- 0x8048000 =

**Stack**

- Return Address:
  - G1
  - 5
  - G2
  - 0x8048000
  - G3
  - ...

**Code**

- G1:
  - pop %eax
  - ret

- G2:
  - pop %ebx
  - ret

- G3:
  - movl %eax, (%ebx)
  - ret
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks"

ROP Example

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
\text{%eax} = 5
\text{%ebx} = 0x8048000
```

```
\text{G1:} \quad \text{pop \ %eax}
\text{ret}

\text{G2:} \quad \text{pop \ %ebx}
\text{ret}

\text{G3:} \quad \text{movl \ %eax, \(%ebx\)}
\text{ret}
```

```
\text{Stack}
\begin{array}{c}
\text{Return Address} \\
\text{G1} \\
\text{5} \\
\text{G2} \\
0x8048000 \\
\text{G3} \\
\text{\ldots} \\
\end{array}
```

```
\text{Registers}
\begin{array}{c}
\text{%eax} = 5 \\
\text{%ebx} = 0x8048000 \\
\end{array}
```

```
\text{Memory}
\text{0x8048000 =}
```
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed "return-oriented attacks" (ROP)

**Example**
- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

```
G1:
  pop %eax
  ret

G2:
  pop %ebx
  ret

G3:
  movl %eax, (%ebx)
  ret
```

Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop %eax</td>
<td>G1</td>
</tr>
<tr>
<td>ret</td>
<td>5</td>
</tr>
<tr>
<td>pop %ebx</td>
<td>G2</td>
</tr>
<tr>
<td>ret</td>
<td>0x8048000</td>
</tr>
<tr>
<td>movl %eax, (%ebx)</td>
<td>G3</td>
</tr>
<tr>
<td>ret</td>
<td>...</td>
</tr>
</tbody>
</table>

Memory

%eax = 5
%ebx = 0x8048000
Code Reuse in General

- Code reuse attacks can be employed more generally to enable adversaries to execute existing code under their control
  - Termed “return-oriented attacks” (ROP)

**Example**

- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000 (without introducing new code)

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: pop %eax</td>
<td>G1</td>
</tr>
<tr>
<td>ret</td>
<td>5</td>
</tr>
<tr>
<td>G2: pop %ebx</td>
<td>G2</td>
</tr>
<tr>
<td>ret</td>
<td>0x8048000</td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx)</td>
<td>G3</td>
</tr>
<tr>
<td>ret</td>
<td>...</td>
</tr>
</tbody>
</table>

Registers

- %eax = 5
- %ebx = 0x8048000
Building ROP Functionality

No-ops

- No-op instruction does nothing but advance %eip
- Return-oriented equivalent:
  - point to return instruction
  - advances %esp
- Useful in nop sled
Building ROP Functionality

Immediate constants

- Instructions can encode constants
- Return-oriented equivalent:
  - Store on the stack;
  - Pop into register to use
Building ROP Functionality

Control flow

- Ordinary programming:
  - (Conditionally) set %eip to new value
- Return-oriented equivalent:
  - (Conditionally) set %esp to new value
Return-oriented Programming

• What can we do with return-oriented programming?
  ‣ Anything any other program can do
  ‣ How do we know?
Return-oriented Programming

• What can we do with return-oriented programming?
  ‣ Anything any other program can do
  ‣ How do we know? Turing completeness

• A language is Turing complete if it has (loosely)
  ‣ Conditional branching
  ‣ Can change memory arbitrarily

• Both are possible with ROP
Finding Gadgets

- Snippets of code ending in “ret” are called gadgets.

- How do we build a complete exploit from available code?
  - Must find the gadgets that are available in that code.

- How do you think one finds all the gadgets in a code region?
Finding Gadgets

• Snippets of code ending in “ret” are called gadgets.

• How do we build a complete exploit from available code?
  ‣ Must find the gadgets that are available in that code.

• How do you think one finds all the gadgets in a code region?
  ‣ From each byte offset in the code region, see what sequence of instructions are encoded until a ”ret” is reached.
  ‣ Find “a, b, c, ret” – where a, b, and c are other instructions.
Finding Gadgets

- Snippets of code ending in “ret” are called gadgets.
- How do we build a complete exploit from available code?
  - Must find the gadgets that are available in that code.
- How do you think one finds all the gadgets in a code region?
  - Start from a “ret” byte “0xc3” at any memory location and work backwards to find the longest useful sequence of instructions for a gadget.
  - Find “a, b, c, ret” – find “c, ret”, then “b, c, ret”, then…
Gadgets and Returns

• Must all useful gadgets end with “ret”?
Gadgets and Returns

• Must all useful gadgets end with “ret”?
  ‣ No, several control transfer functions can be employed to chain gadgets together

• Some examples
  ‣ Jump-oriented programming
  ‣ Call-oriented programming
  ‣ Basic idea – transition to the next gadget through a jump or call rather than using a return
  ‣ So, such attacks are more generally called “code-reuse attacks”
ROP in the Wild

• Do adversaries really employ such attacks?
Gadgets and Returns

- 2010: ROP attacks contained in “exploit packs”
  - Exploit packs are exploits used in penetration testing
- 2013: First ROP-only attack detected
  - Against Adobe Reader XI
  - i.e., no shell code – entire attack within process
- But often there are easier ways to exploit your software flaws
  - Be careful with JIT code – if adversary can modify
  - Why?
Is Code Injection Dead?

- Code Injection Is Still Desirable for Adversaries
  - Add new code for additional attack functionality
    - Could add a new code file and execute
    - But, may still want to use the hijacked process (evade detection)
  - But, given DEP is code injection no longer possible?
Disable DEP

• How would we use code reuse to disable DEP?
• Goal is to allow execution of writable memory (i.e., change page permissions)
  ‣ There’s a system call for that
    int mprotect(void *addr, size_t len, int prot);
  ‣ Sets protection for region of memory starting at address
  ‣ Invoke this system call to allow execution on stack and then start executing from the injected code
Take Away

- Code injection attacks are prevented by **DEP**
  - Also called **W xor X** (write XOR execute)
- But, adversaries can reuse available code in return-oriented programming attacks
  - Generalized to **code-reuse attacks**
- We examined the **ROP mechanism today**
  - That is the one you must know
- Note that ROP (code-reuse) attacks can re-enable the possibility of code injection attacks