CSE543 - Introduction to Computer and Network Security

Module: Return-oriented Programming

Professor Trent Jaeger
Anatomy of Control-Flow Exploits

- Two steps in control-flow exploitation
  - **First** -- attacker gets control of program flow (return address, function pointer)
    - Stack (buffer), heap, format string vulnerability, …
  - **Second** -- attacker uses control of program flow to launch attacks
    - E.g., Code injection
      - Adversary injects malcode into victim
      - E.g., onto stack or into other data region

- How is code injection done?
Code Injection

- Advantage
  - Adversary can install any code they want
    - What code do adversaries want?
  - Defenses
    - **NX bit** - set memory as non-executable (stack)
    - **W (xor) X** - set memory as either writeable or executable, but not both

- What can adversary do to circumvent these defenses and still execute useful code (for them)?
Return-Oriented Programming

• Arbitrary exploitation without code injection

Return-oriented Programming: Exploitation without Code Injection

Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham
University of California, San Diego
Bad code versus bad behavior

“Bad” behavior

Attacker code

“Good” behavior

Application code

Problem: this implication is false!
any sufficiently large program codebase

arbitrary attacker computation and behavior, 
*without* code injection

(in the absence of control-flow integrity)
Return-to-libc

- Divert control flow of exploited program into libc code
  - system(), printf(),
- No code injection required

- Perception of return-into-libc: limited, easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc — remove system()?

- We show: this perception is false.
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)
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

%eax =
%ebx =
```

```
Code

Stack

Registers

Memory

Return Address
```

```
G1
5
G2
0x8048000
G3
...
buf
0x8048000 =
```
ROP Example

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

Code

G1:  
    pop %eax
    ret

G2:  
    pop %ebx
    ret

G3:  
    movl %eax, (%ebx)
    ret

Stack

Registers

%eax = 5
%ebx =

Memory

0x8048000 =

Return Address

buf

G1
5
G2
0x8048000
G3
...
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
```

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax = 5</td>
<td>0x8048000 =</td>
</tr>
<tr>
<td>%ebx =</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>G2</td>
</tr>
<tr>
<td>0x8048000</td>
</tr>
<tr>
<td>G3</td>
</tr>
<tr>
<td>…</td>
</tr>
</tbody>
</table>

Return Address
buf

Code

Memory

%eax = 5
%ebx =
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
```

Registers

- %eax = 5
- %ebx = 0x8048000

Memory

- 0x804800 =

Stack

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

Return Address

buf
ROP Example

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

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

Stack
G1
5
G2
0x8048000
G3
...
Return Address
buf

Registers
%eax = 5
%ebx = 0x8048000

Memory
0x8048000 =
```
ROP Example

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

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

_stack
buf
<table>
<thead>
<tr>
<th>G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>G2</td>
</tr>
<tr>
<td>0x8048000</td>
</tr>
<tr>
<td>G3</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

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

Memory

0x8048000 =
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
```

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

<table>
<thead>
<tr>
<th>Registers</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax = 5</td>
<td>0x8048000 = 5</td>
</tr>
<tr>
<td>%ebx = 0x8048000</td>
<td></td>
</tr>
</tbody>
</table>
```
- 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
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
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
Creating Programs

**Gadgets**: multiple instruction sequences

- Sometimes more than one instruction sequence needed to encode logical unit
- Example: load from memory into register:
  - Load address of source word into %eax
  - Load memory at (%eax) into %ebx
Finding instruction sequences

- Any instruction sequence ending in “ret” is useful — could be part of a gadget

- Algorithmic problem: recover all sequences of valid instructions from libc that end in a “ret” insn

- Idea: at each ret (c3 byte) look back:
  - are preceding $i$ bytes a valid length-i_insn?
  - recurse from found instructions

- Collect instruction sequences in a trie
Conclusions

- Code injection is not necessary for arbitrary exploitation
- Defenses that distinguish “good code” from “bad code” are useless
- Return-oriented programming likely possible on every architecture, not just x86
- Compilers make sophisticated return-oriented exploits easy to write
Advanced Defenses

• Control-flow attack defenses operate at two stages
  ‣ Prevent attacker from getting control
    • StackGuard, heap sanity checks, ASLR, shadow stacks, ...
  ‣ Prevent attacker from using control for malice
    • NX,W (xor) X, ASLR, Control Flow Integrity (CFI), ...

• For maximum security, a system should use a combination of these defenses

• Q. Is subverting control-flow the only goal of an attacker?
Control-Flow Integrity

- **Goal:** Ensure that process control follows source code
  - Adversary can only choose authorized control-flow sequences
- **Build** a model from source code that describes control flow
  - E.g., control-flow graph
- **Enforce** the model on program execution
  - Instrument control-flow code
    - Jumps, calls, returns, ...
- **Challenges**
  - Building accurate model
  - Efficient enforcement
Software Control Flow Integrity

Techniques, Proofs, & Security Applications

Jay Ligatti summer 2004 intern work with:
Úlfar Erlingsson and Martín Abadi
Our Mechanism

call fp

return
Our Mechanism

F_A

call fp

F_B

return

CFG excerpt

A_{call} \rightarrow B_1

A_{call+1} \leftarrow B_{ret}
Our Mechanism

\[ F_A \]

\[ \text{call \ } fp \]

\[ \text{if(*fp != nop IMM}_1\text{) \ halted} \]

\[ F_B \]

\[ \text{nop IMM}_1 \]

\[ \text{return} \]

\[ \text{CFG excerpt} \]

\[ A_{\text{call}} \rightarrow B_1 \]

\[ A_{\text{call+1}} \leftarrow B_{\text{ret}} \]
Our Mechanism

**FA**
- If(*fp != nop IMM₁) halt
- call fp
- nop IMM₂

**FB**
- nop IMM₁
- If(**esp != nop IMM₂) halt
- return

NB: Need to ensure bit patterns for nops appear nowhere else in code memory

CFG excerpt

\[ A_{\text{call}} \rightarrow B_1 \]

\[ A_{\text{call+1}} \leftarrow B_{\text{ret}} \]
More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int → int function.

$F_A$ can call any int → int function

$F_A$ call fp

$F_B$

$F_C$ succ($A_{\text{call}}$) = \{B_1, C_1\}
More Complex CFGs

Maybe statically all we know is that $F_A$ can call any int → int function

Construction: All targets of a computed jump must have the same destination id (IMM) in their nop instruction

$\text{CFG excerpt}$

$A_{\text{call}} \xrightarrow{} B_1 \quad C_1$

$\text{succ}(A_{\text{call}}) = \{B_1, C_1\}$
Imprecise Return Information

Q: What if $F_B$ can return to many functions?

CFG excerpt

$succ(B_{ret}) = \{A_{call+1}, D_{call+1}\}$
Imprecise Return Information

Q: What if $F_B$ can return to many functions?

CFG excerpt

$succ(B_{\text{ret}}) = \{A_{\text{call+1}}, D_{\text{call+1}}\}$

```
if(**esp != nop IMM_2) halt
```

```
call $F_B$
nop IMM_2

return
```
Imprecise Return Information

Q: What if \( F_B \) can return to many functions?
A: Imprecise CFG

 CFG excerpt
\[
\begin{align*}
A_{\text{call+1}} & \rightarrow B_{\text{ret}} \\
D_{\text{call+1}} & \leftarrow B_{\text{ret}} \\
\text{succ}(B_{\text{ret}}) & = \{A_{\text{call+1}}, D_{\text{call+1}}\}
\end{align*}
\]

\( F_A \)

\( F_B \)

call \( F_B \)
nop IMM\(_2\)

\( F_D \)

call \( F_B \)
nop IMM\(_2\)

if(**esp != nop IMM\(_2\)) halt

return

Imprecise Return Information

Q: What if $F_B$ can return to many functions?
A: Imprecise CFG

CFG excerpt

$succ(B_{\text{ret}}) = \{A_{\text{call}+1}, D_{\text{call}+1}\}$

CFG Integrity:
Changes to the PC are only to valid successor PCs, per $succ()$. 

Imprecise CFG

...
No “Zig-Zag” Imprecision

CFG excerpt

A_{call} \rightarrow B_{1} \rightarrow C_{1} \rightarrow E_{call}
No “Zig-Zag” Imprecision

CFG excerpt

\( A_{\text{call}} \rightarrow B_1 \rightarrow C_1 \)

\( E_{\text{call}} \rightarrow C_1 \)
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

CFG excerpt

A_{call} \rightarrow B_1

E_{call} \rightarrow C_1
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

A_{call} \rightarrow B_1

C_1

E_{call}

A_{call} \rightarrow B_1

C_{1A}

E_{call} \rightarrow C_{1E}
More Challenges

- Predicting indirect call targets is hard
- Predicting return targets can be hard
  - Exceptions and signals and setjmp/longjmp
- Runtime generation of indirect jumps
  - E.g., dynamic shared libraries
- Indirect jumps using arithmetic operators
  - E.g., assembly
- *Take away:* CFI is a principled approach to stop control-flow attacks, but challenges remain
Alternatives to CFI?

• What are the fundamental enablers of ROP attacks?
  • **CFI**: violate control flow
  • Adversary can choose gadgets

• Can we prevent adversaries from choosing useful gadgets?
  • In general, adversaries can create/obtain the same binary as is run by the victim
  • But, that need not be the case
Apply Crypto to Code?

- Can we randomize the program’s execution in such a way that an adversary cannot select gadgets?
- Given a secret key and a program address space, encrypt the address space such that:
  - the probability that an adversary can locate a particular instruction (start of gadget) is sufficiently low
  - and the program still runs correctly and efficiently
- Called address space randomization
Prevent Injection on Stack

- One idea applied in practice
- Suppose an adversary wants to inject code onto the stack
  - Write onto the stack (buffer overflow)
  - Jump to that malcode (return address)
- Randomize the base address of the stack on each execution
  - Prevents adversary from predicting malicious return address
- Can we apply this idea more generally?
Prevent Injection on Stack

• One idea applied in practice
• Suppose an adversary wants to inject code onto the stack
  ‣ Write onto the stack (buffer overflow)
  ‣ Jump to that malcode (return address)
• Randomize the base address of the stack on each execution
  ‣ Prevents adversary from predicting malicious return address
• Can we apply this idea more generally?
ASLR

- For control-flow attacks, attacker needs absolute addresses
- Address-space Layout Randomization (ASLR) randomizes base addresses of memory segments on each invocation of the program
  - Attacker cannot predict absolute addresses
- Heap, stack, data, text, mmap, ...
ASLR

• For control-flow attacks, attacker needs absolute addresses
• Address-space Layout Randomization (ASLR) randomizes base addresses of memory segments on each invocation of the program
  ‣ Attacker cannot predict absolute addresses
• Heap, stack, data, text, mmap, ...
ASLR Implementations

• Linux
  ‣ Introduced in Linux 2.6.12 (June 2005)
  ‣ Shacham et al. [2004]: 16 bits of randomization defeated by a (remote) brute force attack in minutes
  ‣ Reality: ASLR for text segment (PIE) is rarely used
    • Only few programs in Linux use PIE
    • Enough gadgets for ROP can be found in unrandomized code [Schwartz 2011]
ASLR Limitations

• Attacks may leak randomization information
  • Disclosure attacks
  • Use buffer over-read to read unauthorized program memory (extract code or randomizing state)
• ASLR can be bypassed by information leaks about memory layout
  ‣ E.g., format string vulnerabilities
• So, what can we do?
  ‣ How do we avoid leaking the “key”?
Conclusion

• Defense against control-flow attacks is an ongoing arms race

• Principled approaches such as CFI and ASLR are promising
  ‣ Significantly raise bar for attackers
  ‣ However, they have implementation limitations
  ‣ Active area of research