CSE543 - Introduction to Computer and Network Security

Module: Advanced Program Vulnerabilities and Defenses

Professor Trent Jaeger
Anatomy of Control-Flow Exploits

- Two steps in control-flow exploitation
- First -- attacker gets control of program flow (return address)
  - Stack (buffer), heap, format string vulnerability
- Second -- attacker uses control of program flow to launch attacks
  - Code injection
    - Defense: NX, W (xor) X
  - return-to-libc
    - Defense: remove unwanted functions (e.g., system)
- How to overcome these limitations???
  - Return-oriented programming
Return-Oriented Programming

• Arbitrary exploitation without code injection or whole-function reuse (return-to-libc)
Bad code versus bad behavior

“Bad” behavior

“Good” behavior

Attacker code

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 Attacks

- Need control of memory around %esp
- Rewrite stack:
  - Buffer overflow on stack
  - Format string vuln to rewrite stack contents
- Move stack:
  - Overwrite saved frame pointer on stack; on leave/ret, move %esp to area under attacker control
  - Overflow function pointer to a register spring for %esp:
    - set or modify %esp from an attacker-controlled register
    - then return
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
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-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
• Use ESP as program counter
  – E.g., Store 5 at address 0x8048000 (without introducing new code)

```
pop %eax
ret

pop %eax
ret

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

- %eax =
- %ebx =

```
G1
5
jmp G2
0x8048000
jump G3
... Return Address
```

buf

```
0x8048000 =
```
ROP Example

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

```
pop %eax
ret
pop %eax
ret
movl %eax, (%ebx)
ret
```

```
Registers
%eax =
%ebx =
```

```
Memory
0x8048000 =
```

```
Code
pop %eax
ret
pop %eax
ret
movl %eax, (%ebx)
ret
```

```
Stack
G1
5
jmp G2
0x8048000
jump G3
...
```

Return Address

buf
ROP Example

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

```
%eax =
%ebx =

pop %eax
ret
pop %eax
ret
movl %eax, (%ebx)
ret
```

```
stack
G1
5
jmp G2
0x8048000
jump G3
...
```

```
memory
Return Address
```

registrs

```
0x8048000 =
```
ROP Example

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

```assembly
pop %eax
ret

pop %eax
ret

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
</table>
| pop %eax
ret | G1
5   |
| pop %eax
ret | 5
jmp G2 |
| movl %eax, (%ebx)
ret | 0x8048000 |
| | jump G3 |
| | ... |
| | Return Address |
| | buf |

Registers

%eax = 5
%ebx =

Memory

0x8048000 =
ROP Example

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

```plaintext
pop %eax
ret
pop %eax
ret
movl %eax, (%ebx)
ret

%eax = 5
%ebx =

G1
5
jmp G2
0x8048000
jump G3
...

Return Address
buf

0x8048000 =
```
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>pop %eax</td>
<td>G1</td>
</tr>
<tr>
<td>ret</td>
<td>5</td>
</tr>
<tr>
<td>pop %eax</td>
<td>jmp G2</td>
</tr>
<tr>
<td>ret</td>
<td>0x8048000</td>
</tr>
<tr>
<td>movl %eax, (%ebx)</td>
<td>jump G3</td>
</tr>
<tr>
<td>ret</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 =</td>
</tr>
<tr>
<td>%ebx = 0x8048000</td>
<td></td>
</tr>
</tbody>
</table>
• Use ESP as program counter
  – E.g., Store 5 at address 0x8048000 (without introducing new code)

```
Code
pop %eax
ret
pop %eax
ret
movl %eax, (%ebx)
ret
```

```
Stack
G1
5
jmp G2
0x8048000
jump G3
...
```

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

```
Memory
0x8048000 =
```

```
Return Address
buf
```
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>pop %eax</td>
<td>G1</td>
</tr>
<tr>
<td>ret</td>
<td>5</td>
</tr>
<tr>
<td>pop %eax</td>
<td>jmp G2</td>
</tr>
<tr>
<td>ret</td>
<td>0x8048000</td>
</tr>
<tr>
<td>movl %eax, (%ebx)</td>
<td>jump G3</td>
</tr>
<tr>
<td>ret</td>
<td>...</td>
</tr>
</tbody>
</table>

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

- Memory
  - 0x8048000 = 5
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

\[ F_A \]

\[ \text{call } f_p \]
\[ \text{nop } \text{IMM}_2 \]

\[ F_B \]

\[ \text{nop } \text{IMM}_1 \]

\[ \text{if}(\ast f_p \neq \text{nop } \text{IMM}_1) \text{ halt} \]

\[ \text{if}(\ast \ast \text{esp} \neq \text{nop } \text{IMM}_2) \text{ halt} \]

\[ \text{return} \]

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 $\text{int} \rightarrow \text{int}$ function.

\begin{align*}
F_A & \xleftarrow{\text{call}} F_A \\
\text{if}(*fp \neq \text{nop IMM}_1) & \text{halt} \\
\text{call} fp & \xleftarrow{\text{nop IMM}} F_B \\
& \xleftarrow{\text{nop IMM}} F_C
\end{align*}

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

CFG excerpt:

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

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

CFG excerpt:

$A_{\text{call}+1} \rightarrow B_{\text{ret}}$

$D_{\text{call}+1} \rightarrow B_{\text{ret}}$

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

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

```
if(**esp != nop IMM2) halt
return
```
No “Zig-Zag” Imprecision

Solution I: Allow the imprecision

Solution II: Duplicate code to remove zig-zags

CFG excerpt

A_{call} \rightarrow B_1

E_{call} \rightarrow C_1

CFG excerpt

A_{call} \rightarrow B_1

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

• Returns used as jumps
  ‣ E.g., signal handling

• Exceptions

• 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
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, ...

Stack ➔ Heap ➔ Data ➔ Text
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]

• Windows
  ‣ Introduced from Vista onwards (Jan 2007)
  ‣ Reality: Only few programs opt in for ASLR
    • E.g., Oracle’s Java JRE, Adobe Reader, Mozilla Firefox, and Apple Quicktime (or one of their libraries) are not marked ASLR-compatible

• ASLR can be bypassed by information leaks about memory layout
  ‣ E.g., format string vulnerabilities
Conclusion

• Defense against control-flow and data attacks is an ongoing arms race
• Principled approaches such as CFI and ASLR are promising
  ‣ Significantly raised bar for attackers
  ‣ However, they have implementation limitations
  ‣ Active area of research