Return-oriented Programming

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

```
_start:
  call main

int main() {
  char buf[8];
  gets(buf);
  printf("You typed: %s", buf);
}
```
Buffer Overflow

```c
_start:
    call main

int main() {
    char buf[8];
    gets(buf);
    printf("You typed: %s", buf);
}
```

EIP

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0804321</td>
<td>0x0804321</td>
</tr>
<tr>
<td>0x0804480</td>
<td>0x0804480</td>
</tr>
<tr>
<td>0x0804484</td>
<td>0x0804484</td>
</tr>
<tr>
<td>0xbf000000</td>
<td>0xbf000000</td>
</tr>
<tr>
<td>0xbf000004</td>
<td>0xbf000004</td>
</tr>
<tr>
<td>0xbf000008</td>
<td>0xbf000008</td>
</tr>
<tr>
<td>0xbf00000b</td>
<td>0xbf00000b</td>
</tr>
</tbody>
</table>

ESP

- `buf`: Pointer to the buffer where the user input is stored.
Buffer Overflow

```c
_start:
    call main

int main() {
    char buf[8];
    gets(buf);
    printf("You typed: \%s\", buf);
}
```
Buffer Overflow

```
#include <stdio.h>

int main() {
    char buf[8];
    gets(buf);
    printf("You typed: \%s\", buf);
    return 0;
}
```

EIP: 0x0804480

ESP: 0xbf000000

Return Address: 0x80484321

```
EIP: 0x12345678 0x90abcdef 0xbf000004
```

Return Address:
```
0x80484321
```

buf:
```
0x12345678
```

```
buf
```

```
0xbf000000
0xbf000004
0xbf000008
0xbf00000b
```
Buffer Overflow

```
_start:
0x0804321   call main

int main() {
  char buf[8];
  gets(buf);
  printf("You typed: %s", buf);
}
```

EIP

0x0804480     gets(buf);
0x12345678 0x90abcdef 0xbf000004

ESP

0xbf000000 0x80484321
0xbf000004 0x90abcdef
0xbf000008 0x12345678
0xbf00000b buf

...
Buffer Overflow

```c
_start:
    call main

int main() {
    char buf[8];
    gets(buf);
    printf("You typed: %s", buf);
}
```

EIP

```
0x0804321 0x0804480 0x0804484
    call main
    gets(buf);
```

ESP

```
0xbf000000 0xbf000004 0x12345678 0x90abcdef 0xbf000008 0x12345678
    buf
    0x0804484
    printf("You typed: %s", buf);
```

Return Address

```
0x0804321
```

0x0804480

g`ets(buf);`
Buffer Overflow

```c
int main() {
    char buf[8];
    gets(buf);
    printf("You typed: %s", buf);
}
```

EIP

```
0x0804321 _start:
    call main
```

ESP

```
0xbf000000 0xbf000004
0xbf000004 0x90abcdef
0xbf000008 0x12345678
0xbf00000b...
```

Return Address

```
0x12345678 0x90abcdef 0xbf000004
```
Buffer Overflow

```c
int main() {
    char buf[8];
    gets(buf);
    printf("You typed: %s", buf);
}
```

EIP: 0x0804321

Return Address: 0xbf000004

ESP:
- 0xbf000000
- 0xbf000004
- 0xbf000008
- 0xbf00000b

buf:
- 0x12345678
- 0x90abcdef
- 0xbf000004

...
Buffer Overflow

```
_start:
call main

int main() {
    char buf[8];
    gets(buf);
    printf("You typed: %s", buf);
}
```

EIP

ESP

Return Address

buf

0x0804321 call main

0x0804480

0x0804321 call main

0x0804484

0x0804480 gets(buf);

0x12345678 0x90abcdef 0xbf000004

printf("You typed: %s", buf);

0x12345678 0x90abcdef 0xbf000004

0xbf000000 0xbf000004 0xbf000008 0xbf00000b
Buffer Overflow Defense

- **W xor X**
  - Pages marked write can’t be executed
- **Return-to-libc**

```
0xbf000000
0xbf000004
0xbf000008
0xbf00000b

<table>
<thead>
<tr>
<th>_libc_system</th>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>/bin/sh</td>
<td>buf</td>
</tr>
<tr>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
```
ASLR

- Randomize bases of memory regions
  - Stack (Thwarts traditional stack overflow)
  - Mmap (Thwarts return-to-libc)
  - Brk (Heap – Thwarts traditional heap overflow)
  - Exec (Program binary)

- Not enabled by default

```
0xbf??????
0xbf??????
0xbf??????
0xbf??????
0xbf??????

<table>
<thead>
<tr>
<th>_libc_system</th>
<th>/bin/sh</th>
<th>-c</th>
<th>Return Address ???</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>buf</td>
</tr>
</tbody>
</table>

...
Anatomy of Control Flow Attacks

• Two steps

• First, the attacker changes the control flow of the program
  ‣ In buffer overflow, overwrite the return address on the stack
  ‣ What are the ways that this can be done?

• Second, the attacker uses this change to run code of their choice
  ‣ In buffer overflow, inject code on stack
  ‣ What are the ways that this can be done?
Return-oriented Programming

• General approach to control flow attacks

• Demonstrates how general the two steps of a control flow attack can be

• First, change program control flow
  ▸ In any way

• Then, run any code of attackers’ choosing, including the code in the existing program
• Use ESP as program counter
  ▷ E.g., Store 5 at address 0x8048000
  • without introducing new code

```
ROP

- 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 %ebx</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 =
%ebx =

Memory
0x8048000 =

Return Address
buf
```
- Use ESP as program counter
  - E.g., Store 5 at address 0x8048000
    - without introducing new code

```
ROP

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

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

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

Memory
0x8048000 =

Registers
%eax =
%ebx =
```

Buf
Return Address
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code

```
 Systems and Internet Infrastructure Security (SIIS) Laboratory

ROP

%eax = 5
%ebx =

```

```
Code

pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret

```

```
Stack

G1
5
jmp G2
0x8048000
jump G3
...

Registers

%eax = 5
%ebx =

Memory

0x8048000 =

```

Return Address

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

```
%eax = 5
%ebx = 0x8048000 = Registers
movl %eax, (%ebx) ret

pop %eax
ret
pop %ebx
ret
```

![Diagram showing code and stack operations]

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

Memory
0x8048000 =
```

Return Address
buf
• Use ESP as program counter
  ‣ E.g., Store 5 at address 0x8048000
  • without introducing new code

ROP

Code

pop %eax
ret

pop %ebx
ret

movl %eax, (%ebx)
ret

Stack

G1
5
jmp G2
0x8048000
jump G3
...

Return Address

buf

Registers

%eax = 5

%ebx = 0x8048000

Memory

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

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

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

```
pop %eax
ret

pop %ebx
ret
```

![Diagram](image)
ROP

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

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

Memory
- 0x8048000 = 5

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

Stack
- G1
- 5
- jmp G2
- 0x8048000
- jump G3
- ...
- Return Address
- buf
Return-oriented Programming

• How can an adversary make this happen?

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

"Good" behavior

Attacker code

Application code

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

arbitrary attacker computation and behavior, \textit{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
- 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
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 instructions 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
Works on non-x86 Systems

Return-oriented programming on SPARC

- Use Solaris 10 libc: 1.3 MB
- New techniques:
  - Use instruction sequences that are *suffixes* of real functions
  - Dataflow within a gadget:
    - Use structured dataflow to dovetail with calling convention
  - Dataflow between gadgets:
    - Each gadget is memory-memory
- Turing-complete computation!

**Conjecture**: Return-oriented programming likely possible on *every* architecture.
Works on non-x86 Systems

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
Summary

- The types of attacks that we must defend against are becoming more complex
- Return-oriented programming shows us that \textit{any} attacker-dictated change in program control flow can lead to arbitrary malice
- Stuxnet shows that ad hoc system defenses can be evaded by an adversary
- We must apply principled approaches to defense to make significant strides in defense