CMPSC 497
More on Overflow Vulnerabilities

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

- Despite knowledge of buffer overflows for over 40 years, they have not been eliminated
- This is partly due to the wide variety of exploit options
  - **Variety of targets**: can exploit more than return addresses – any code addresses or data
  - **Variety of uses**: can exploit on read and write
  - **Variety of exploits**: can inject or reuse code
  - **Variety of workarounds**: current defenses are incomplete
Available Defenses

• Available defenses do not prevent all options
  ‣ **Stackguard**: A canary before return address on stack
  ‣ **DEP or W xor X**: Stack memory is not executable

• This combination of defenses prevents the classic buffer overflow attack, but there are many options

• Plus, both defenses may be disabled by other attacks
StackGuard Limitations

- **Obvious limitation**: only protects the return address
  - What about other local variables?

```c
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}
if (authenticated)
    ProcessPacket(packet);
```
Function Initialization: Stacks

- Packet overflows overwrite the authenticated value

```
<table>
<thead>
<tr>
<th>packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>authenticated</td>
</tr>
<tr>
<td>old ebp</td>
</tr>
<tr>
<td><strong>CANARY</strong></td>
</tr>
<tr>
<td>ret</td>
</tr>
<tr>
<td>stack frame</td>
</tr>
</tbody>
</table>
```
StackGuard Limitations

• **Obvious limitation**: only protects the return address
  
  ‣ What is the exploit?

```
int authenticated = 0;
char packet[1000];

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}

if (authenticated)
    ProcessPacket(packet);
```
StackGuard Limitations

- **Obvious limitation:** only protects the return address
  - What about other local variables?
  - Of course, there are also other data on the stack that may also require protection
    - Code addresses – function pointers
    - Data that impacts control flow – as in example
    - Data that may impact operation of program
- Any ways to address this limitation?
StackGuard Limitations

- **Big limitation**: Disclosure attacks
  - By performing a buffer “overread”
  - Example is the famous *Heartbleed attack* against SSL
  - Why is this a problem for Stackguard canaries?

```c
char packet[10];
...
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len2);
```
StackGuard Limitations

• **Big limitation**: Disclosure attacks
  ‣ By performing a buffer “overread”
  ‣ Example is the famous Heartbleed attack against SSL
  ‣ Why is this a problem for Stackguard?

```c
char packet[10];
...
// suppose len is adversary controlled
strncpy(buf, packet, len);
send(fd, buf, len);
```

![Diagram](image)

StackGuard Limitations

- **Big limitation**: disclosure attacks
  - By performing a buffer “overread”
  - One may extract the canary values by reading beyond the end of stack buffers
  - Which would enable the adversary to learn the (supposedly secret) canary value
  - How would you exploit this?
DEP Limitations

- **Big limitation**: code injection is not necessary to construct adversary-controlled exploit code
  - Code reuse attacks
Vs. Injecting Shell Code

- 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
```

Code Reuse

• How can we invoke execve without code injection?
Code Reuse

• 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
    • Reusing available code to do what the adversary wants
How can we invoke `execve` without code injection?

- Use the code directly

The difference is subtle, but significant

```
stack frame for main

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

```
buffer
execve "/bin/sh"
2
stack frame for main
```
Code Reuse

- How would we use code reuse to disable DEP?
- Goal is to allow execution of stack memory
Code Reuse

• How would we use code reuse to disable DEP?
• Goal is to allow execution of stack memory
  ‣ 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
Heap Overflows

- Another region of memory that may be vulnerable to overflows is heap memory
  - A buffer overflow of a buffer allocated on the heap is called a heap overflow

```c
int authenticated = 0;
char *packet = (char *)malloc(1000);

while (!authenticated) {
    PacketRead(packet);
    if (Authenticate(packet))
        authenticated = 1;
}
if (authenticated)
    ProcessPacket(packet);
```
Heap Overflows

- Overflows on heap also possible
  ```c
  char *packet = malloc(1000)
  ptr[1000] = ‘M’;
  ```
- “Classical” heap overflow corrupts metadata
  - Heap metadata maintains chunk size, previous and next pointers, ...
    - Heap metadata is `inline` with heap data
    - And waits for heap management functions (`malloc`, `free`) to write corrupted metadata to target locations
Heap Overflows

- Heap allocators maintain a doubly-linked list of allocated and free chunks
- `malloc()` and `free()` modify this list

http://www.sans.edu/student-files/presentations/heap_overflows_notes.pdf
Heap Overflows

- free() removes a chunk from allocated list

\[
\text{chunk2} \rightarrow \text{bk} \rightarrow \text{fd} = \text{chunk2} \rightarrow \text{fd} \\
\text{chunk2} \rightarrow \text{fd} \rightarrow \text{bk} = \text{chunk2} \rightarrow \text{bk}
\]

- By overflowing chunk2, attacker controls \( \text{bk} \) and \( \text{fd} \)
  - Controls both where and what data is written!
  - Arbitrarily change memory (e.g., function pointers)
Heap Overflows

• By overflowing chunk2, attacker controls bk and fd
  ‣ Controls both where and what data is written!
    • Assign chunk2->fd to value to want to write
    • Assign chunk2->bk to address X (where you want to write)
      • Less an offset of the fd field in the structure
• Free() removes a chunk from allocated list
  chunk2->bk->fd = chunk2->fd
  chunk2->fd->bk = chunk2->bk

• What’s the result?
Heap Overflows

• By overflowing chunk2, attacker controls bk and fd
  ‣ Controls both where and what data is written!
    • Assign chunk2->fd to value to want to write
    • Assign chunk2->bk to address X (where you want to write)
      • Less an offset of the fd field in the structure
  • Free() removes a chunk from allocated list
    chunk2->bk->fd = chunk2->fd
    addrX->fd = value
    chunk2->fd->bk = chunk2->bk
    value->bk = addrX

• What’s the result?
  • Change a memory address to a new pointer value (in data)
Defenses

• What would you suggest for defenses to prevent heap overflows?
  ‣ Heap canaries?
  ‣ DEP or W xor X?
Defenses

- What would you suggest for defenses to prevent heap overflows?
  - Heap canaries?
    - Heap canaries would be numerous
    - And prone to disclosure attacks still
  - DEP?
    - Prevents you from assigning a code pointer to an address directly – value = addr will fail
    - But can replace function pointers, cause they are data
    - Code reuse is still possible through heap overflows
Heap Overflow Defenses

• Separate data and metadata
  ‣ e.g., OpenBSD’s allocator (Variation of PHKmalloc)

• Sanity checks during heap management
  
  ```
  free(chunk2) -->
  assert(chunk2->fd->bk == chunk2)
  assert(chunk2->bk->fd == chunk2)
  ```

  ‣ Added to GNU libc 2.3.5

• Randomization

• Q. What are analogous defenses for stack overflows?
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”
    - by Hovav Shacham and his colleagues
    - Next few slides are Prof Shacham’s
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"
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 Thesis

any sufficiently large program codebase

arbitrary attacker computation and behavior, \textit{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)
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"

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
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
```

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

*Registers*

- %eax =
- %ebx =

*Memory*

- 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)

\[
\begin{align*}
\text{G1:} & \quad \text{pop } \%\text{eax} \\
& \quad \text{ret} \\
\text{G2:} & \quad \text{pop } \%\text{ebx} \\
& \quad \text{ret} \\
\text{G3:} & \quad \text{movl } \%\text{eax}, (\%\text{ebx}) \\
& \quad \text{ret}
\end{align*}
\]

\[
\begin{align*}
\text{ Registers } & \quad \%\text{eax} = 5 \\
& \quad \%\text{ebx} = \\
\text{ Memory } & \quad 0x8048000 = 5 \\
& \quad \text{buf}
\end{align*}
\]

\[
\begin{align*}
\text{ Code } & \quad \text{pop } \%\text{eax} \\
& \quad \text{ret} \\
\text{ Stack } & \quad \text{G1} \\
& \quad 5 \\
& \quad \text{G2} \\
& \quad 0x8048000 \\
& \quad \text{G3} \\
& \quad \ldots
\end{align*}
\]
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 ret</td>
<td>G1</td>
</tr>
<tr>
<td>G2: pop %ebx ret</td>
<td>5</td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx) ret</td>
<td>G2</td>
</tr>
<tr>
<td></td>
<td>0x80480000</td>
</tr>
<tr>
<td></td>
<td>G3</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Registers

- %eax = 5

Memory

- 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
```

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

```
Registers
Memory
```

```
<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1: pop %eax, ret</td>
<td>G1</td>
</tr>
<tr>
<td>G2: pop %ebx, ret</td>
<td>5</td>
</tr>
<tr>
<td>G3: movl %eax, (%ebx), ret</td>
<td>G2</td>
</tr>
<tr>
<td></td>
<td>0x80480000</td>
</tr>
<tr>
<td></td>
<td>G3</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Return Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
</tr>
</tbody>
</table>
```

```
%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:</td>
<td>pop %eax</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td>G2:</td>
<td>pop %ebx</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
<tr>
<td>G3:</td>
<td>movl %eax, (%ebx)</td>
</tr>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

Registers

- %eax = 5
- %ebx = 0x8048000

Memory

- 0x8048000 = buf

Diagram:

- G1
- G2
- G3
- 5
- 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)

Code

<table>
<thead>
<tr>
<th>G1:</th>
<th>pop %eax</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G2:</th>
<th>pop %ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>G3:</th>
<th>movl %eax, (%ebx)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ret</td>
</tr>
</tbody>
</table>

Stack

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

Return Address

buf

Registers

%eax = 5
%ebx = 0x8048000

Memory

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)

```plaintext
G1:  pop %eax
    ret

G2:  pop %ebx
    ret

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

**Stack**

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

- Memory
  - 0x8048000 = 5

- Return Address
  - buf

Diagram:

- Code: G1, G2, G3
- Stack: G1, G2, G3
- Registers: %eax = 5, %ebx = 0x8048000
- Memory: 0x8048000 = 5
- Return Address:
  - 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"

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
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”

Building ROP Functionality

Immediate constants

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

Building ROP Functionality

- 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 in ROP
Take Away

- Buffer overflows provide a wide variety of options for adversaries, depending on the software flaw.

- The result is that defenses that have become widely available are only effective against a fraction of the possible exploit options.
  - And can be disabled by some types of exploits.

- We discussed variants of the classic buffer overflow attack.
  - Disclosure attacks, core reuse attacks, heap overflows, and return-oriented attacks.