CMPSC 447
Buffer Overflow Vulnerabilities

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

- Early example of a method to exploit a “memory error” in a C program
- Discovered in the 1970s
- Leveraged by the Morris Worm in 1988 – first large-scale exploit
- Leveraged by subsequent attacks in the early 2000s that led to security rethink
- Still a problem today – Check out CVEs for “buffer overflow”
Memory Error

• A **memory error** allows a program statement to access memory outside of that allocated for the variables processed in the statement

• Common case: **Buffer overflow**
  - The C language allows writes to memory addresses specified by pointers
    - char buf[10] – *buf* can be used as a pointer
  - C functions enable writing based on the size of the input or a length value
    - *strcpy* and *strncpy*
  - However, **does not ensure** writes only within the buffer
Morris Worm

- Robert Morris, a 23-year old Cornell PhD student
  - Wrote a small (99 line) program
  - Launched on November 3, 1988
  - Simply disabled the Internet
- Used a buffer overflow in a program called `fingerd`
  - To get adversary-controlled code running
- Then spread to other hosts – cracked passwords and leveraged open LAN configurations
- Covered its tracks (set is own process name to `sh`, prevented accurate cores, re-forked itself)
Process Address Space

- **Text**: static code
- **Data**: also called heap
  - static variables
  - dynamically allocated data (malloc, new)
- **Stack**: program execution stacks
Program Stack

- For implementing procedure calls and returns
- Keep track of program execution and state by storing
  - local variables
  - arguments to the called procedure (callee)
  - return address of the calling procedure (caller)
  - ...
Program Stack

Stack Segment

The stack supports nested invocation calls.

Information pushed on the stack as a result of a function call is called a frame.

```java
b() {...}
a() {
    b();
}
main() {
    a();
}
```

Low memory
- Unallocated
- Stack frame for `b()`
- Stack frame for `a()`
- Stack frame for `main()`

High memory

A stack frame is created for each subroutine and destroyed upon return.

*Slide by Robert Seacord*
Stack Frames

- Stack grows **from high mem to low mem addresses**
- The **stack pointer** points to the current “top of the stack” – last thing pushed on the stack (that matters)
  - ESP in Intel architectures
- The **frame pointer** points to the start of the current frame
  - also called the base pointer
    - EBP in Intel architectures
- The stack is modified during
  - function calls, function prologue, function epilogue and operations on stack variables (locals and args)
A Running Example

```c
void function(int a, int b) {
   char buffer[12];
   gets(buffer);
   return;
}

void main() {
   int x;
   x = 0;
   function(1, 2);
   x = 1;
   printf("%d\n", x);
}
```

Run “gcc –S –o example.s example.c” to see its assembly code
# Function Calls

function (1,2)

<table>
<thead>
<tr>
<th>pushl $2</th>
<th>push the 2\textsuperscript{nd} arg to stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl $1</td>
<td>push the 1\textsuperscript{st} arg to stack</td>
</tr>
<tr>
<td>call function</td>
<td>push the ret addr onto the stack,</td>
</tr>
<tr>
<td></td>
<td>and jumps to the function</td>
</tr>
</tbody>
</table>
Function Calls: Stacks

Before

After

esp → stack frame for main

esp →

ret

1

2

stack frame for main

ebp →

ebp →
Function Initialization

```c
void function(int a, int b) {
    pushl %ebp
    movl %esp, %ebp
    subl $12, %esp
    // saves the prior frame pointer
    // sets the new frame pointer
    // allocate space for local variables
}
```

Function prologue
Function Initialization: Stacks

Before

After

esp →

ebp →

ret

1

2

stack frame for main

buffer

old ebp

ret

1

2

stack frame for main
Function Return

function return;

<table>
<thead>
<tr>
<th>movl %ebp, %esp</th>
<th>restores the old stack pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>popl %ebp</td>
<td>restores the prior frame pointer</td>
</tr>
<tr>
<td>ret</td>
<td>gets the return address at current stack pointer, and jumps to it</td>
</tr>
</tbody>
</table>

Function epilogue
Function Return: Stacks

Before

<table>
<thead>
<tr>
<th>esp</th>
<th>buffer</th>
<th>old ebp</th>
<th>ret</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td>stack frame for main</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>esp</th>
<th>buffer</th>
<th>old ebp</th>
<th>ret</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebp</td>
<td>stack frame for main</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Return to Calling Function

In main again – following return…

<table>
<thead>
<tr>
<th>pushl $2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushl $1</td>
</tr>
<tr>
<td>call function</td>
</tr>
<tr>
<td>addl $8, %esp</td>
</tr>
</tbody>
</table>

restores the stack pointer for caller
Return to Calling Function: Stacks

Before

esp

ebp

buffer

old ebp

ret

1

2

stack frame for main

After

esp

ebp

buffer

old ebp

ret

1

2

stack frame for main
A Running Example

```c
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n", x);
}
```

![Stack frame diagram]

- `esp` points to the top of the stack frame for `main`.
- `ebp` points to the old `ebp` value.
- `buffer` is the buffer allocated on the stack.
- `ret` indicates the return address.
- The numbers 1 and 2 represent the positions of ebp and esp, respectively.
void function(int a, int b) {
    char buffer[12];
    gets(buffer);

    int* ret = (int *)buffer + 2;
    *ret = ?;

    return;
}
void function(int a, int b) {
    char buffer[12];
    gets(buffer);

    int* ret = (int*) buffer+16;
    *ret = *ret + 1;    // assuming one-byte store

    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}

The output will be 0

the original return address
the new return address
Previous Attack

• Not very realistic
  ‣ Attackers are usually not allowed to modify code
  ‣ Threat model: the only thing they can affect is the input
  ‣ Can they still carry out similar attacks?
    • **YES**, because of possible buffer overflows
Buffer Overflows

- A **buffer overflow** occurs when data is written outside of the boundaries of the memory allocated to a particular data structure (buffer)
- Happens when buffer boundaries are neglected and unchecked
- Can be exploited to modify memory after buffer
  - Stack: *return address*, local variables, *function pointers*, etc.
  - Heap: data structures and metadata (next time)
- Also, a **buffer underflow** to modify memory prior
Smashing the Stack

- Occurs when a buffer overflow overwrites other data in the program stack.
- Successful exploits can overwrite the return address on the stack enabling the execution of arbitrary code on the targeted machine.
- What happens if we input a large string?
  - ./example
    - ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
- Segmentation fault – why is that?
What Happened?

```c
void function(int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}
```

If the input is large, then `gets(buffer)` will write outside the bound of buffer, and the return address is overwritten – with “ffff” (in ASCII), which likely is not a legal code address – seg fault.
void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n", x);
}
void function (int a, int b) {
    char buffer[12];
    gets(buffer);
    return;
}

void main() {
    int x;
    x = 0;
    function(1,2);
    x = 1;
    printf("%d\n",x);
}
Code Injection

- Attacker creates a malicious argument—a specially crafted string that contains a pointer to malicious code provided by the attacker.
- When the function returns, control is transferred to the malicious code.
  - Injected code runs with the permission of the vulnerable program when the function returns.
  - Programs running as root or other elevated privileges are normally targeted.
- Programs with the setuid bit on.
Injecting Shell Code

execve ("/bin/sh")

ret

1

2

stack frame for main

- This brings up a shell (logical view – real later)
- Adversary can execute any command in the shell
- The shell has the same privilege as the process
- Often, a process with the root privilege is attacked
Injecting Shell Code

- How do you invoke “execve” using injected code?

![Diagram of stack memory]

Figure 6.5: Buffer overflow of stack-based local variable.

An overflow of buffer var3 overwrites higher memory, including:
- return addr
- old FP
- var1
- var2
- var3

- The overwritten return address may point back into injected code or to any other address.

- SP

- Increasing addresses

- NO-OP SLED: Among several challenges in crafting injected code for stack execution, one is: precisely predicting the target transfer address that the to-be-executed code will end up at, and within this same injected input, including that target address at a location that will overwrite the stack frame’s return address. To reduce the precision needed to compute an exact target address, a common tactic is to precede the to-be-executed code by a sequence of machine code NO-OP (no-operation) instructions. This is called a no-op sled.

Transferring control anywhere within the sled results in execution of the code sequence beginning at the end of the sled. Since the presence of a NO-OP sled is a telltale sign of an attack, attackers may replace literal NO-OP instructions with equivalent instructions having no effect (e.g., OR 0 to a register). This complicates sled discovery.

6.4 Heap-based buffer overflows and heap spraying

Beyond the stack, overflows may affect buffers in heap memory and the data segment (BSS and Data in Fig. 6.3). Traditionally, many systems have left the heap and BSS not only writable (necessary), but also executable (unnecessary, dangerous). The data

This term may make more sense to readers familiar with bobcats or snow toboggans, which continue sliding down a hill to its bottom (the code to be executed).
Injecting Shell Code

• Inject the address of the “execve” function at the return address or elsewhere in stack reference by the return address

  ‣ “execve” is a function in libc that is dynamically linked into the process address space

• To invoke a function in a library it must be able to find that address itself as well

• How is that done? Your program calls “execve” thru a stub (procedure linkage table), which retrieves the address set at link time (in the global offset table)
Injecting Shell Code

• Example of PLT code (from `objdump -dl`)

0x08048730 <execve@plt>:

8048730:  ff 25 1c d1 04 08  jmp  *0x804d11c
8048736:  68 28 00 00 00  push  $0x28
804873b:  e9 90 ff ff ff  jmp  80486d0

0x08048740 <strncpy@plt>:

8048740:  ff 25 20 d1 04 08  jmp  *0x804d120
8048746:  68 30 00 00 00  push  $0x30
804874b:  e9 80 ff ff ff  jmp  80486d0
## Injecting Shell Code

<table>
<thead>
<tr>
<th>Stack Frame for main</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr of /bin/sh</td>
</tr>
<tr>
<td>execve@plt</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

- Overwrite return address with address of code to run next (e.g., `execve@plt`)
  - What address?
- Provide argument(s) above – pointer to “/bin/sh” command
  - Where to put it?
- And then “null” for last arg (env)
Any C(++) code acting on untrusted input is at risk

- Code taking input over untrusted network
  - E.g., sendmail, web browser, wireless network driver,...

- Code taking input from untrusted user on multi-user system,
  - esp. services running with high privileges (as ROOT on Unix/Linux, as SYSTEM on Windows)

- Code processing untrusted files
  - that have been downloaded or emailed

- Also embedded software, e.g., in devices with (wireless) network connection such as mobile phones with Bluetooth, wireless smartcards in new passport or OV card, airplane navigation systems, ...
Take Away

- **Memory errors** enable processes to write to memory outside the expectation range
- The classic example is the **buffer overflow**, which is still a common attack vector today
- A buffer overflow vulnerability allows an adversary to overwrite the memory beyond the buffer on the stack
  - But runtime state is also on the stack – return address
- We discussed methods to inject and reuse code
- Available defenses are not complete