BUFFE
R
CS33 PROJECT GUT UP
You will exploit a program using buffer overflow attacks to deliver unintended results.

This type of exploit/cyber attack is one of the oldest and most dangerous, effective and persistent type of attacks because it relies on how computers fundamentally work.
ROADMAP

- **cs0330_install buffer**
- Get your unique cookie according to handout directions. This is crucial to your project having a unique solution.
- **$./makecookie <cslogin>**
- Read and understand **every** section of the handout and the x86_64 guide before you start.
- Go through the levels one at a time (they increase in order of difficulty).
THE STACK

- **High Memory Address**
- **Buffer Start**
- **Low Memory Address**
  - **Stack Grows**
  - **Bottom (origin) of stack**
  - **Buffer grows toward the bottom (higher address)**
  - **Top of stack**
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

Stack frame for caller
```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

Stack frame for caller
THE STACK

```c
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

THE STACK
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

Stack frame for caller

rip
rbp
rsp
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
# THE STACK

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```

```c
int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
```c
int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
THE STACK

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
THE STACK

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
THE STACK

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Get(buf);
    return 1;
}
```
THE STACK

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
THE STACK - REGISTERS

%rbp: points to the base of the current stack frame

%rip: points to the next instruction

%rsp: points to the top of the stack
THE STACK - INSTRUCTIONS

**push x**: stores the value x in memory location %rsp, decrements %rsp

**pop x**: stores the value stored in %rsp in x, increments %rsp

**jmp <addr>**: updates %rip to <addr> so that the next instruction executed is <addr>

**call <addr>**: gives program control to callee function at <addr>

- **push %rip**: pushes the next instruction pointer (the return address) onto the stack
- **jmp <addr>**: then jumps to the location indicated by <addr>

**ret**: returns control back to caller function

- **pop %rip**: pops return address off of the stack and into the instruction pointer
BUFFER OVERFLOW?

```c
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
BUFFER OVERFLOW?

```c
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```

Stack frame for `caller`
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

Stack frame for caller
BUFFER OVERFLOW?

```c
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```

Stack frame for `caller`

- Return address
- Old rbp value for `caller`
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf()
{
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```

Stack frame for `caller`

Old rbp value for `caller`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>

rbp
rsp
rip
**BUFFER OVERFLOW?**

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```

Stack frame for `caller`

```
0x12345678 <hacked>
00000000
00000000 00000000
00000000 00000000
00000000 00000000
00000000
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```

Stack frame for `caller`

```
0x12345678 <hacked>
00000000
```
BUFFER OVERFLOW?

```c
int getbuf()
{
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int caller()
{
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```
```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {  
    int a = 2;  
    int b = 3;  
    int c = getbuf();  
    printf("%d", c);  
}

int getbuf() {  
    char buf[32];  
    // reads input into buf  
    Gets(buf);  
    return 1;  
}

int hacked() {  
    printf("what is below C level?\n");  
    printf("assembly!\n");  
    return 1;  
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    gets(buf);
    return 1;
}

int hacked() {
    printf("what is below C level?\n");
    printf("assembly!\n")
    return 1;
}
```
BUFFER OVERFLOW?

```c
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

int getbuf() {
    char buf[32];
    // reads input into buf
    Gets(buf);
    return 1;
}

int hacked() {
    printf("what is below C level?\n");
    printf("assembly!\n")
    return 1;
}
```
BUFFER OVERFLOW?

- Stack only allocates x number of bytes for the char array.
- Overflow the char array by reading more than x characters from stdin.
- Use this to change the return address!

Stack frame for caller

Return address

Old rbp value for caller

Local variables for callee

Gets(buf);

Stack frame for caller

0x12345678 <hacked>

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000

00000000 00000000 00000000 00000000
ENDIANNESS

- The order of bytes in a word
- Varies by architecture
- Big Endian: most significant byte first
- Little Endian: least significant byte first
  - Department machines are little endian
- Different from bit order
DEM

Generating Machine Code
DEMO

$ cat exploit | ./hex2raw > exploit-raw.txt
$ ./buffer -u <cslogin> < exploit-raw.txt

$ gdb buffer
$ run -u <cslogin> < exploit-raw.txt
TIPS

- Understand how the stack works.
- Understand what a buffer overflow is
- GDB disassemble command: `disas` or `layout asm`
- GDB provides info about registers: `info registers`
- `si` is a GDB command which you can use to step over a single x86-64 instruction
MORE TIPS

- Remember that dept. machines are little-endian
  - Hex addresses will be in reverse order (addresses increase as you go down the exploit file)
- `hex2raw` expects two-digit hex values separated by whitespace
  - To create a byte with a value of 0, you need to specify 00.
- Use newlines to separate functionality and to make your code readable
- Comment each section with what you are doing (important)
  - You can use /* this */ to comment inline
GOOD REFERENCES

(Can also be found on CS33 website)

- http://cs.brown.edu/courses/csci0330/docs/guides/x64_cheatsheet.pdf
WHAT YOU’LL LEARN

● How to exploit a buffer overflow vulnerability
● How to avoid buffer overflow vulnerabilities (not trivial)!
● Reading and writing assembly code
● x86 function call conventions
● How buffer overflow attacks take advantage of said x86-64 function call conventions
QUESTIONS?