
BUFFE
R
CS33 PROJECT

PROJECT OVERVIEW

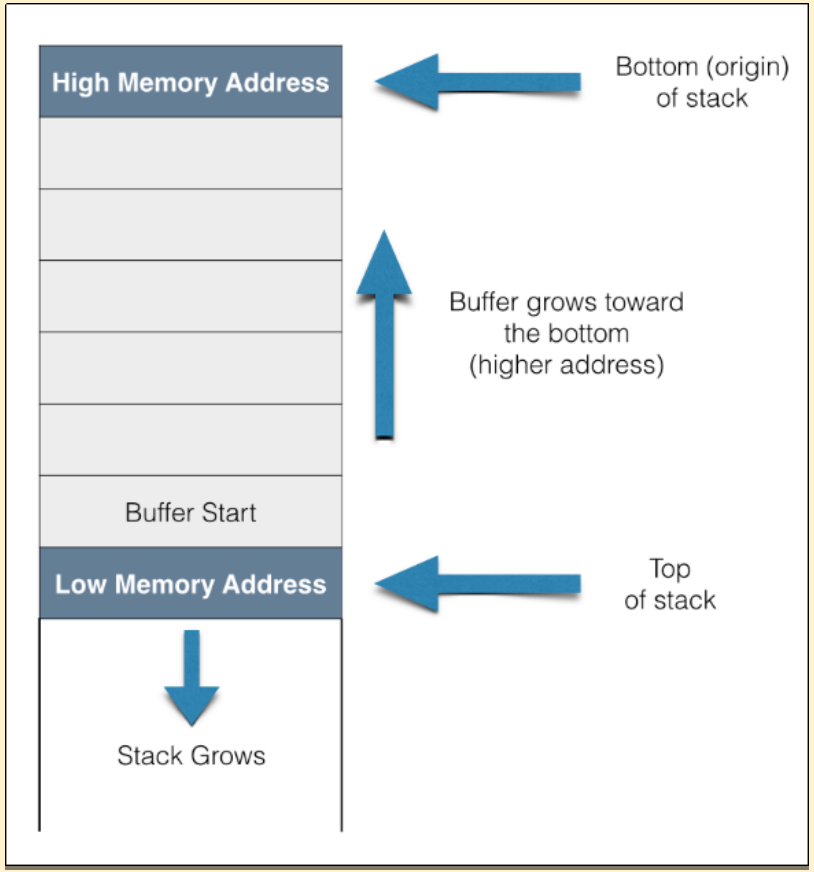
You will exploit a program using buffer overflow attacks to deliver unintended results.

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

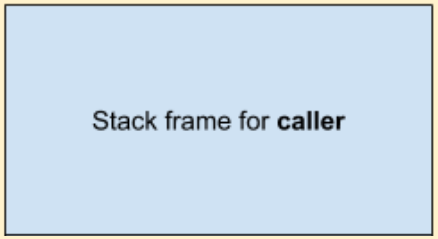

High Memory Address
Bottom (origin) of stack
Buffer grows toward the bottom (higher address)
Buffer Start
Low Memory Address
Top of stack
Stack Grows

THE STACK

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

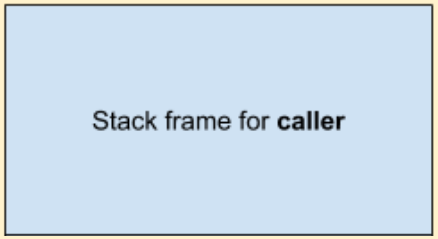
THE STACK

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



THE STACK

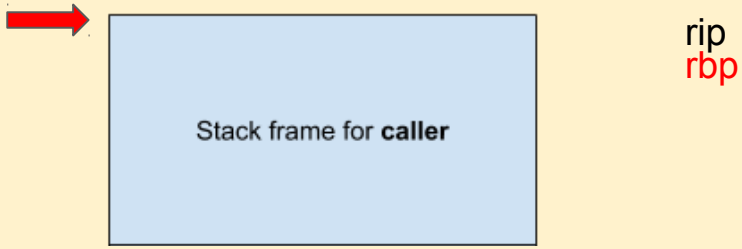
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip

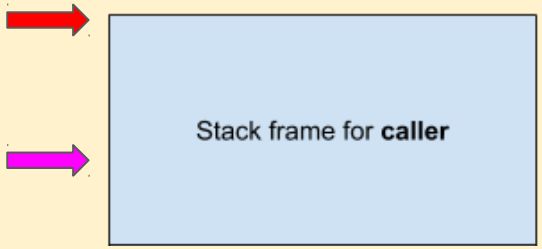
THE STACK

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



THE STACK

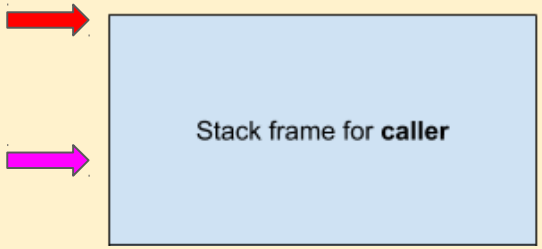
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

THE STACK

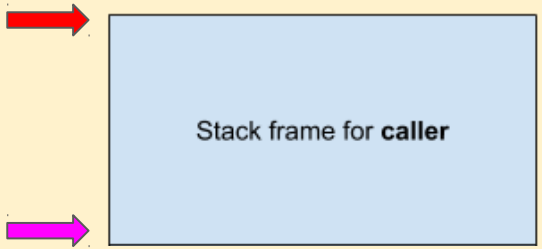
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

THE STACK

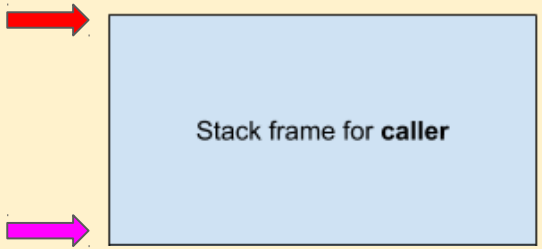
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

THE STACK

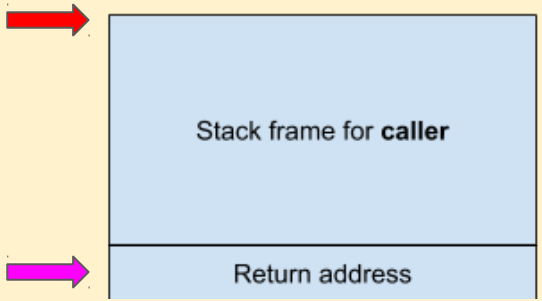
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

THE STACK

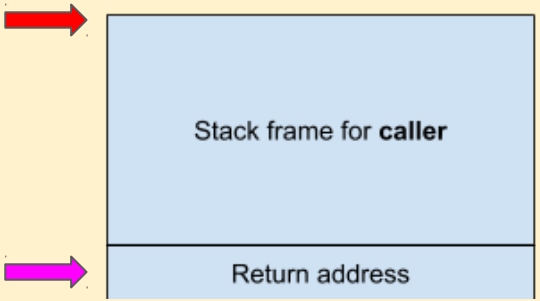
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

THE STACK

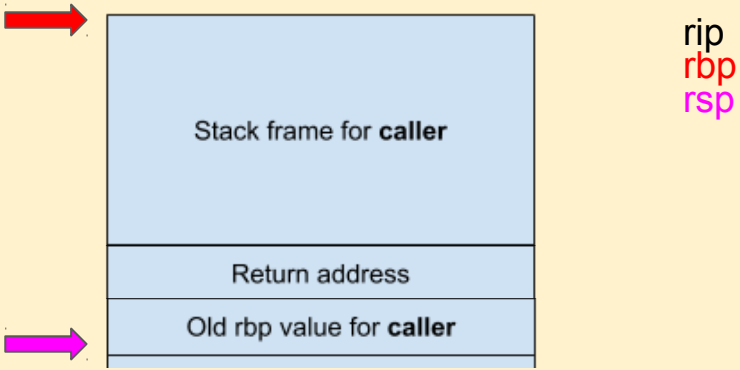
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

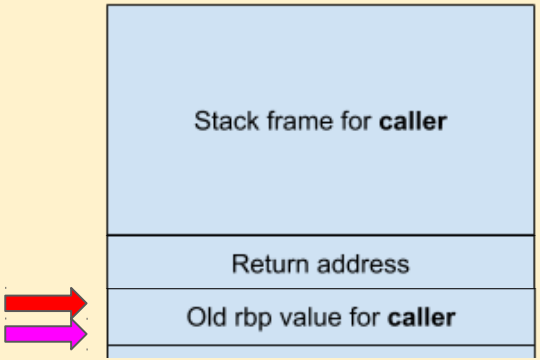
THE STACK

```
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

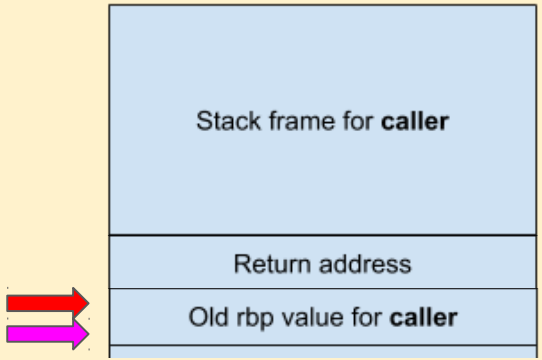
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

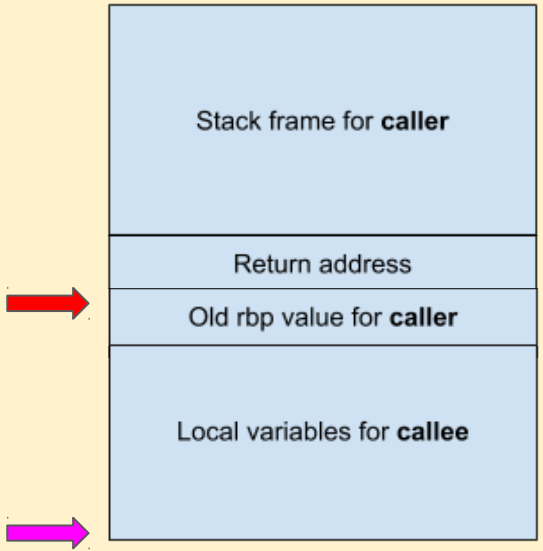
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

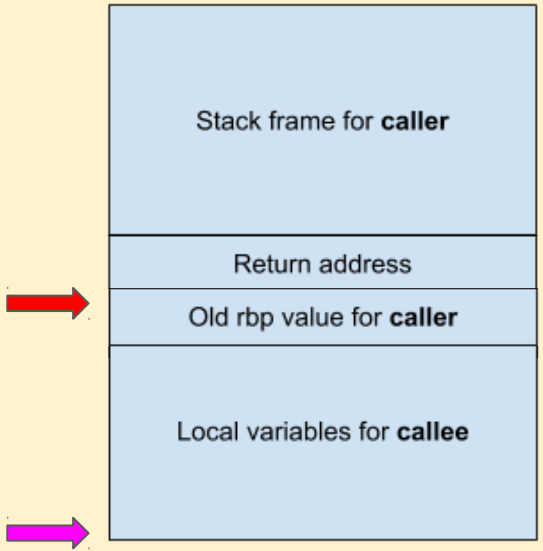
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

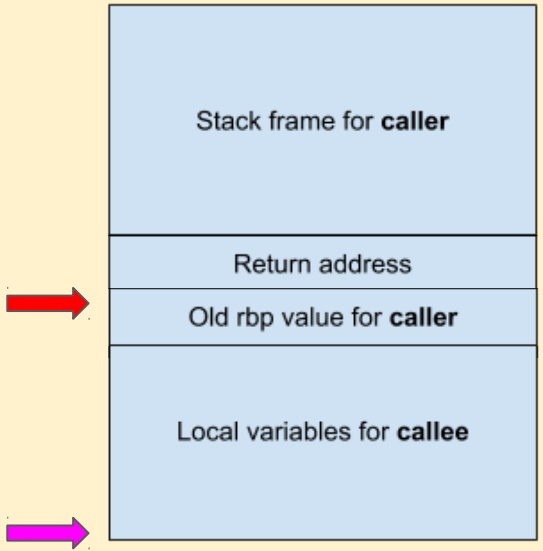
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

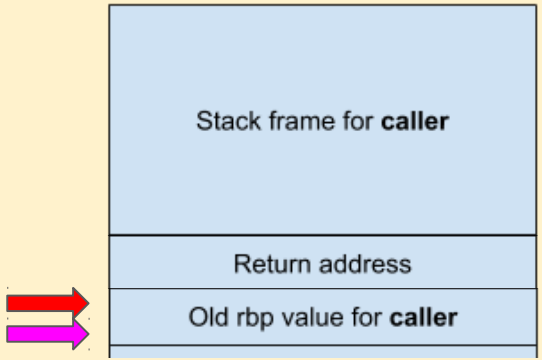
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

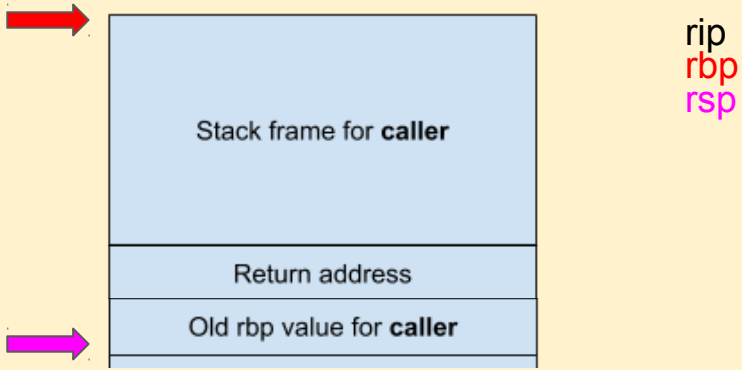
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

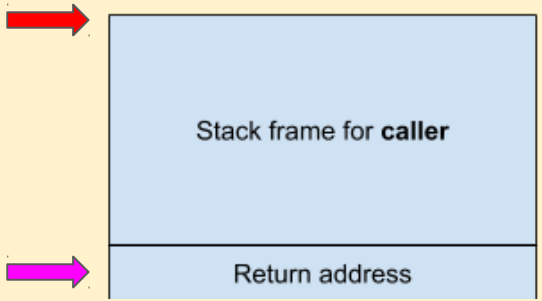
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

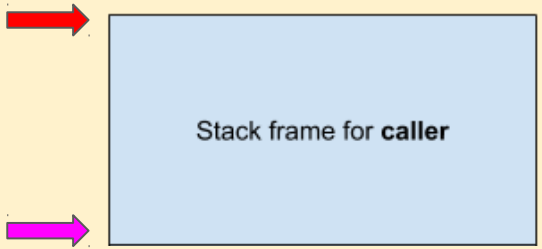
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

THE STACK

```
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;
}
```



rip
rbp
rsp

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

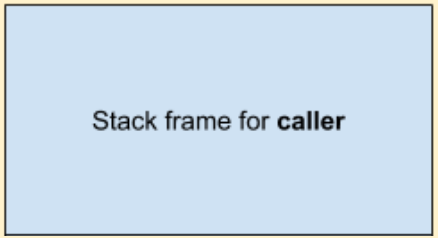
- **pop %rip**: pops return address off of the stack and into the instruction pointer

BUFFER OVERFLOW?

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

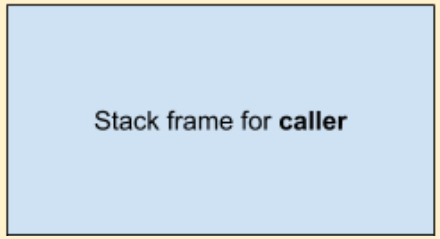
BUFFER OVERFLOW?

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



BUFFER OVERFLOW?

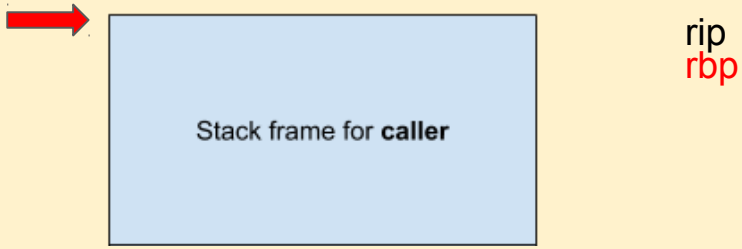
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip

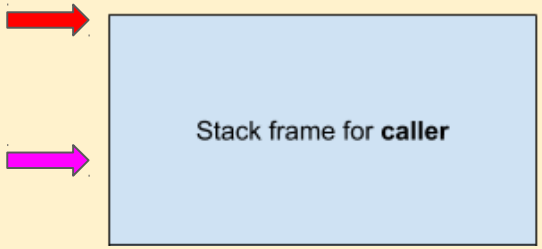
BUFFER OVERFLOW?

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



BUFFER OVERFLOW?

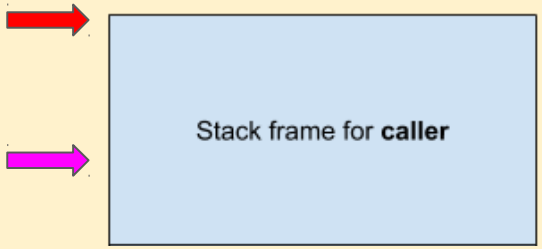
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

BUFFER OVERFLOW?

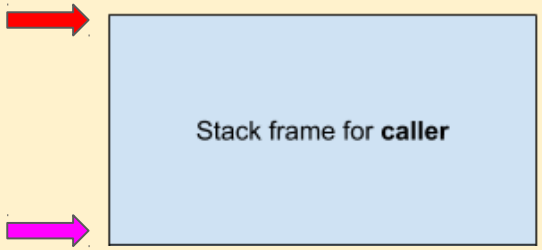
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

BUFFER OVERFLOW?

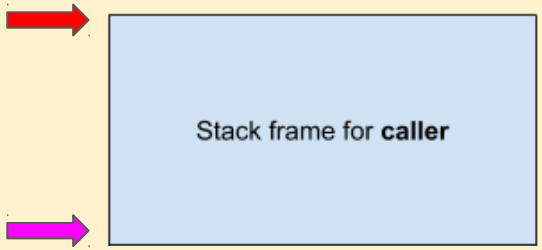
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

BUFFER OVERFLOW?

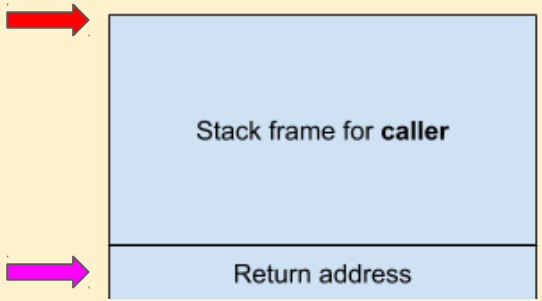
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

BUFFER OVERFLOW?

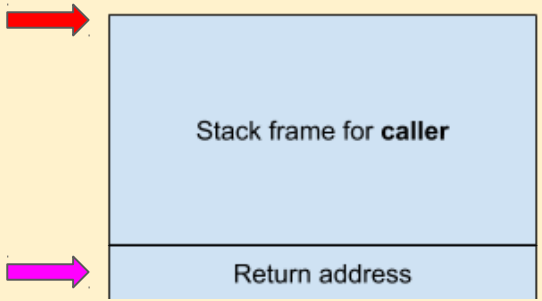
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}
```



rip
rbp
rsp

BUFFER OVERFLOW?

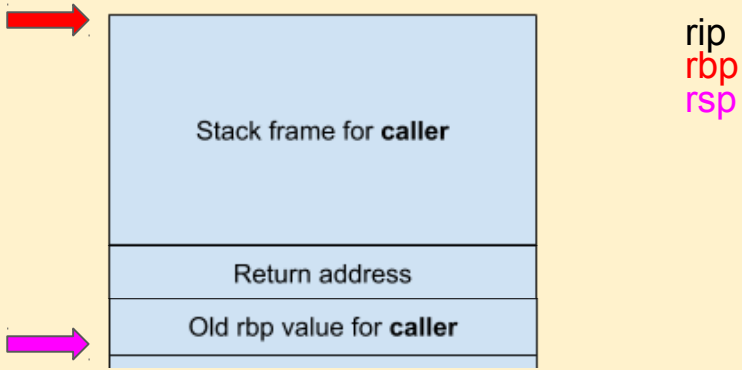
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

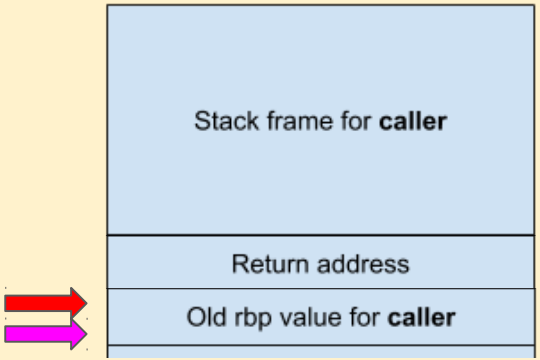
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

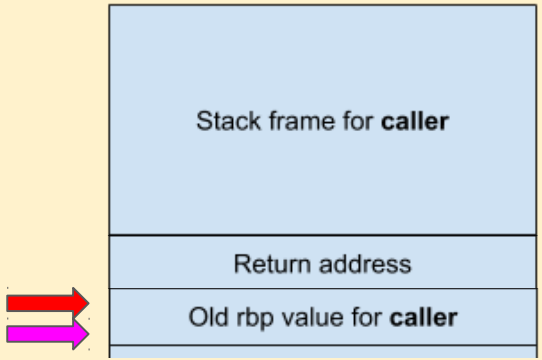
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

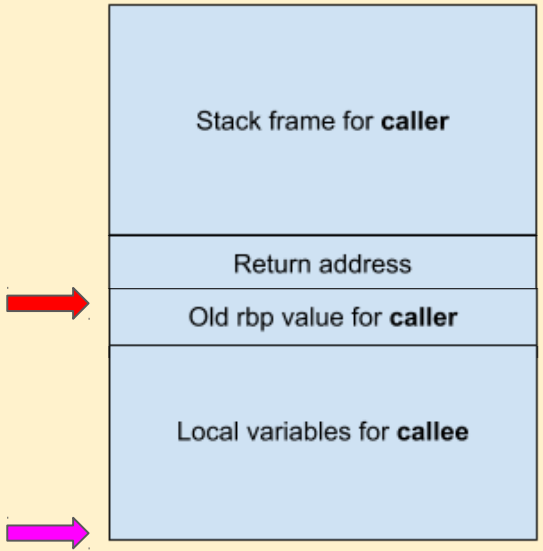
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

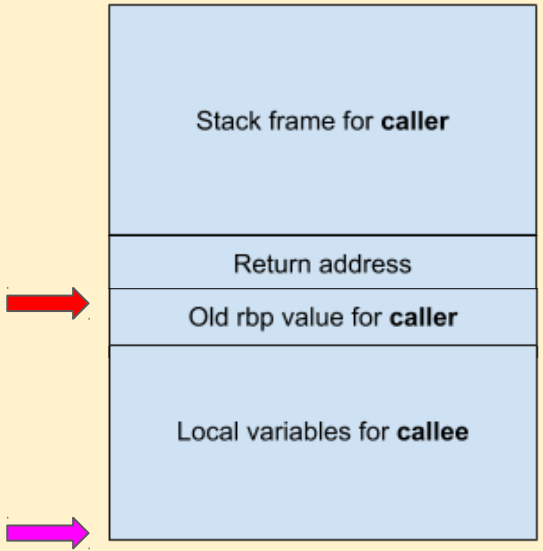
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

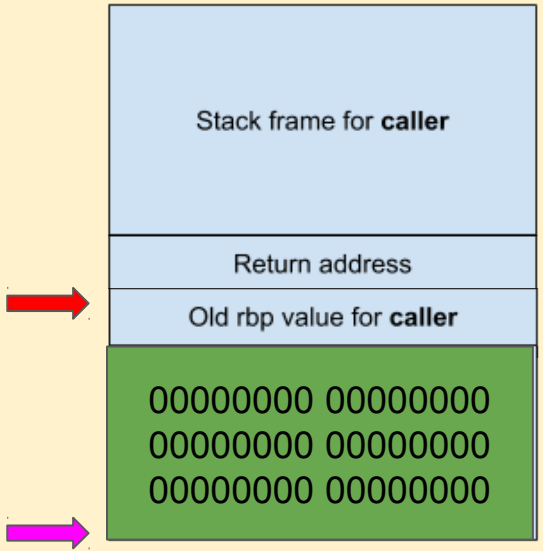
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

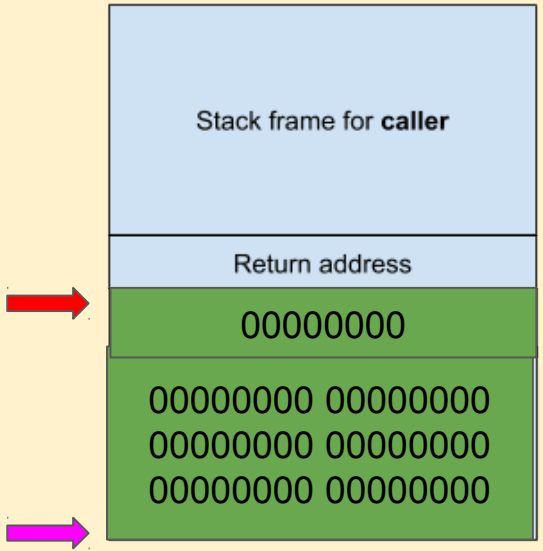
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

BUFFER OVERFLOW?

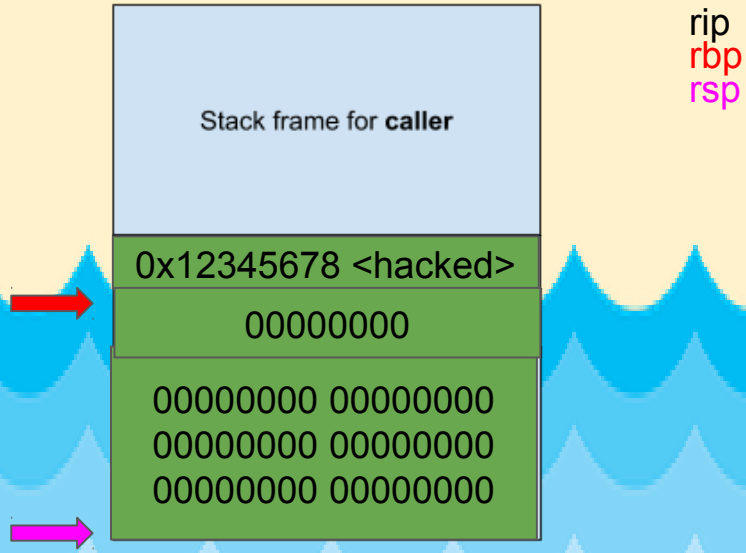
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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



rip
rbp
rsp

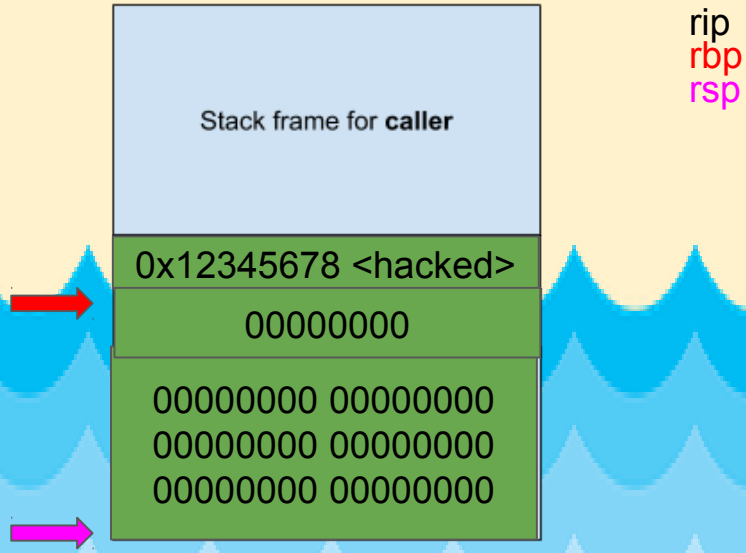
BUFFER OVERFLOW?

```
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;
}
```



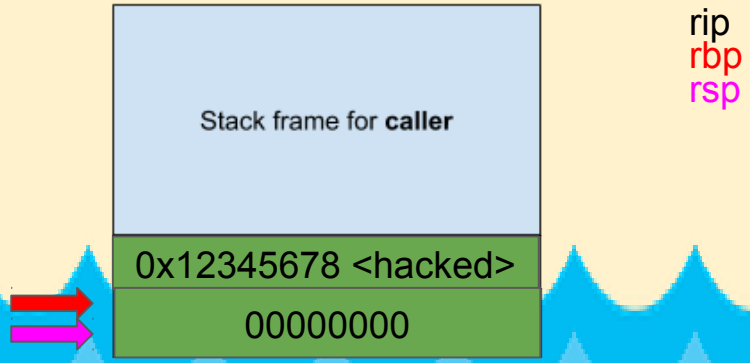
BUFFER OVERFLOW?

```
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;
}
```



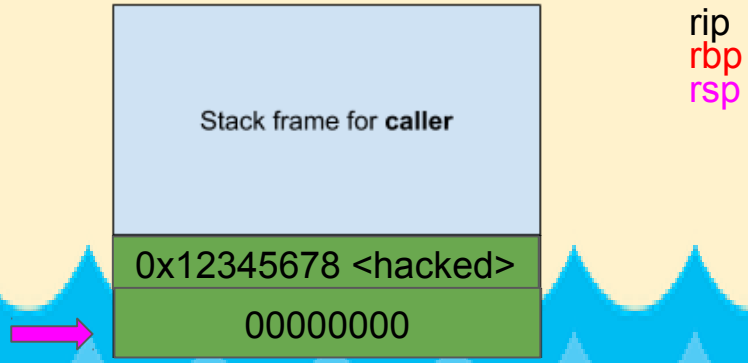
BUFFER OVERFLOW?

```
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;
}
```



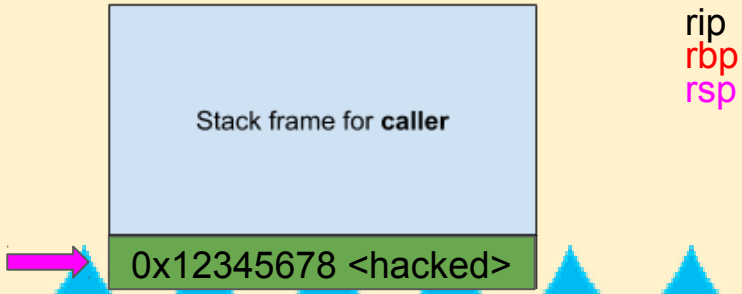
BUFFER OVERFLOW?

```
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;
}
```



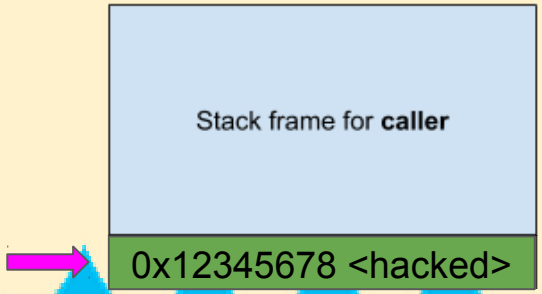
BUFFER OVERFLOW?

```
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?

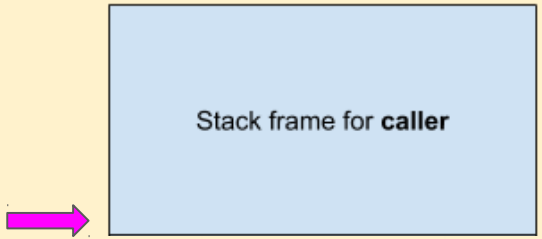
```
int caller() {
    int a = 2;
    int b = 3;
    int c = getbuf();
    printf("%d", c);
}

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?

```
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;
}
```

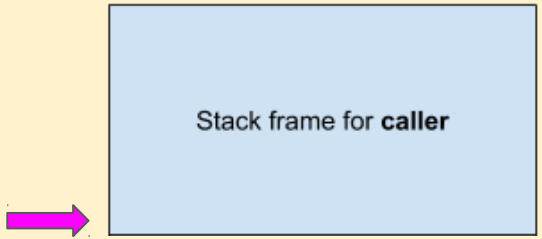


rip
rbp
rsp

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

BUFFER OVERFLOW?

```
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;
}
```

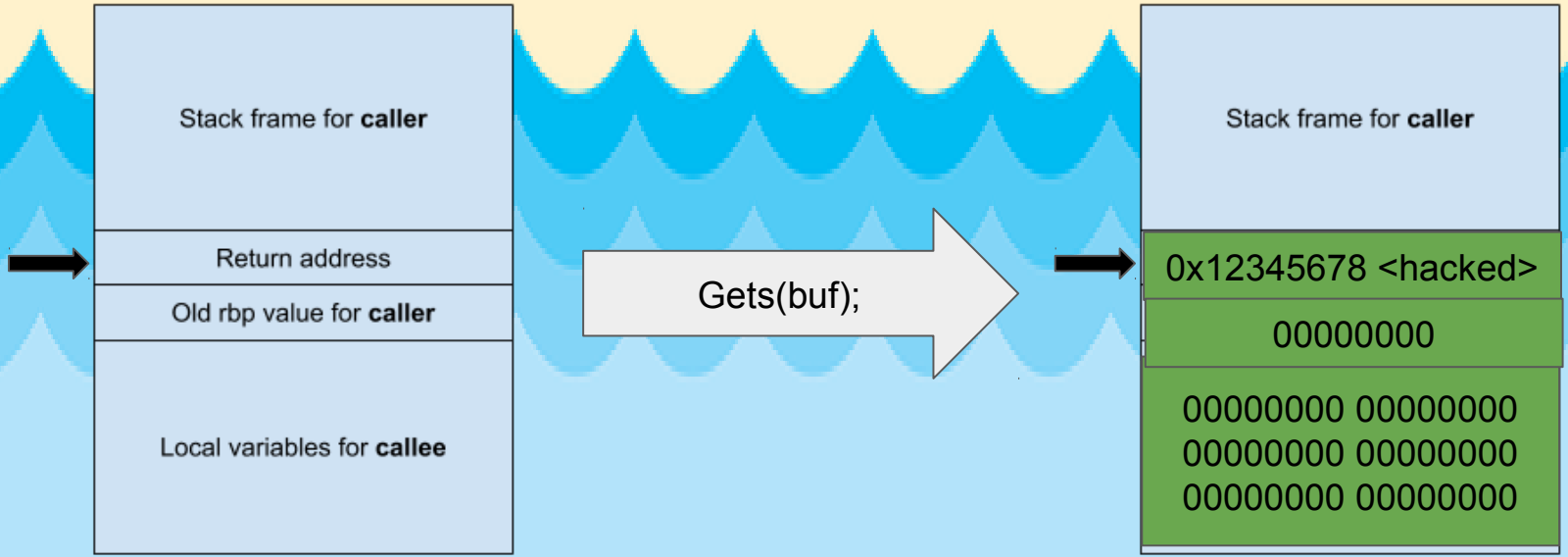


rip
rbp
rsp

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

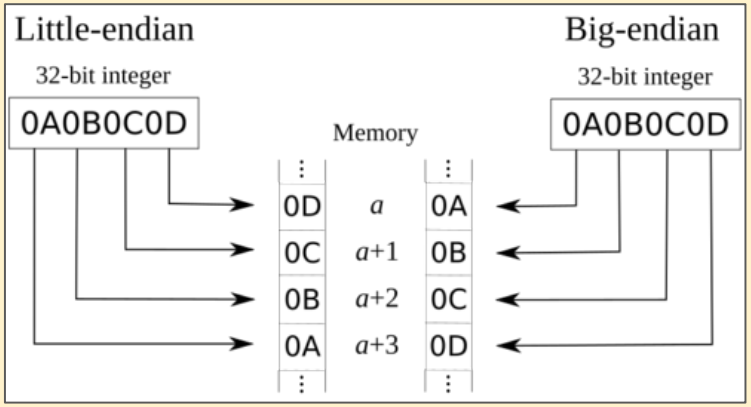
BUFFER OVERFLOW?

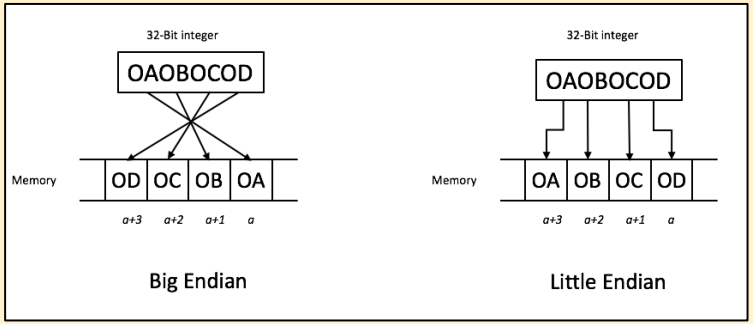
- Stack only allocates x number of bytes for the char array.
- Overflow the char array by reading more than x characters from stdin
- Use this to change the return address!



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





DEMO

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:

  (gdb) info registers

  (gdb) i r
- **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
- http://cs.brown.edu/courses/csci0330/docs/guides/gdb.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?