# COMPSCI 390R

## Buffer Overflows & Stack Exploits

# Topics to Cover:

1. Project 1 due 2/28 at midnight
   I’m hosting office hours today, 4:30-6pm LGRT 212

2. Homework 2 releasing today
   Due in 1 week

# Recap of what we've learned so far:

1. CS 230 Review
    a) ELF file structure
    b) x86-64 ASM

2. Reverse Engineering w/ Ghidra

3. Code Auditing
    a) Integer overflow/underflows
    b) Type Conversion
    c) Sizeof errors

# Recap of what we've learned so far:

So far we've found a lot of ways to corrupt the stack memory, but what can we actually do with that?

Segmentation Faults cause crashes but that doesn't get us much besides a Denial of Service attack, we want more

- Denial of Service attacks are still important to report, 1836 CVE's were DoS attacks in 2021, making up 16.3% of all attacks

# Let's Get Hacking!

# Corrupting the Stack:

Lets examine our code

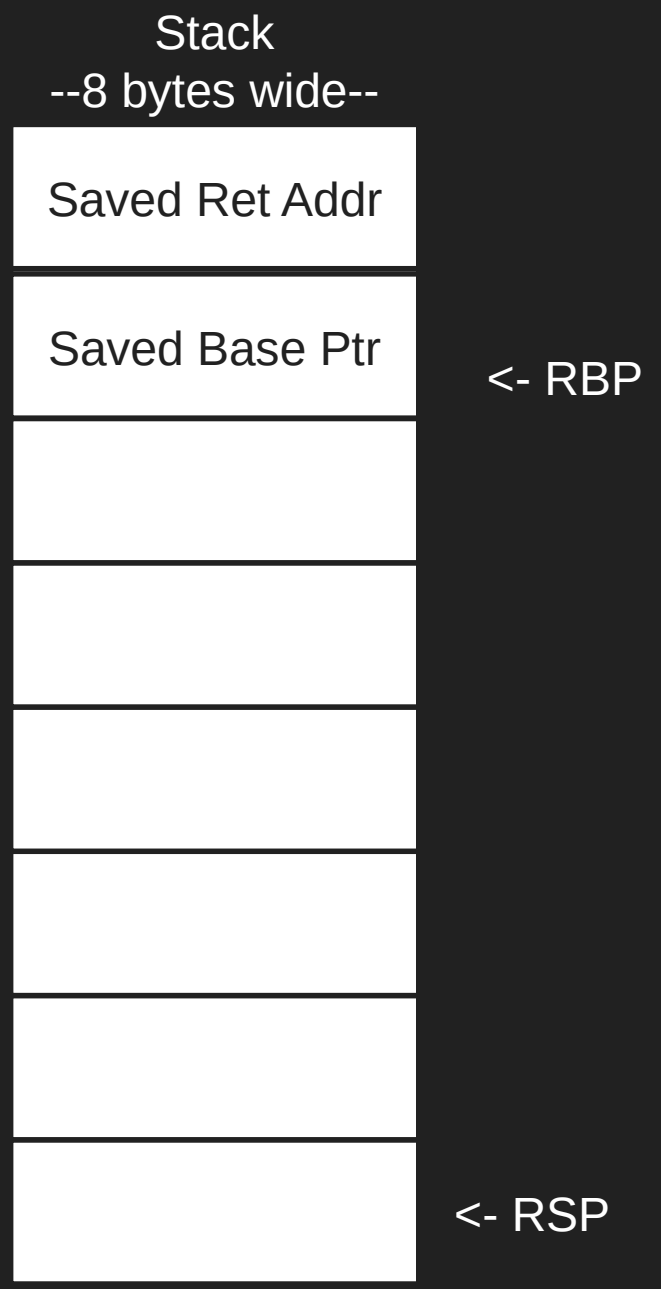
```
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

# Corrupting the Stack:

## Lets examine our code

```c
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame

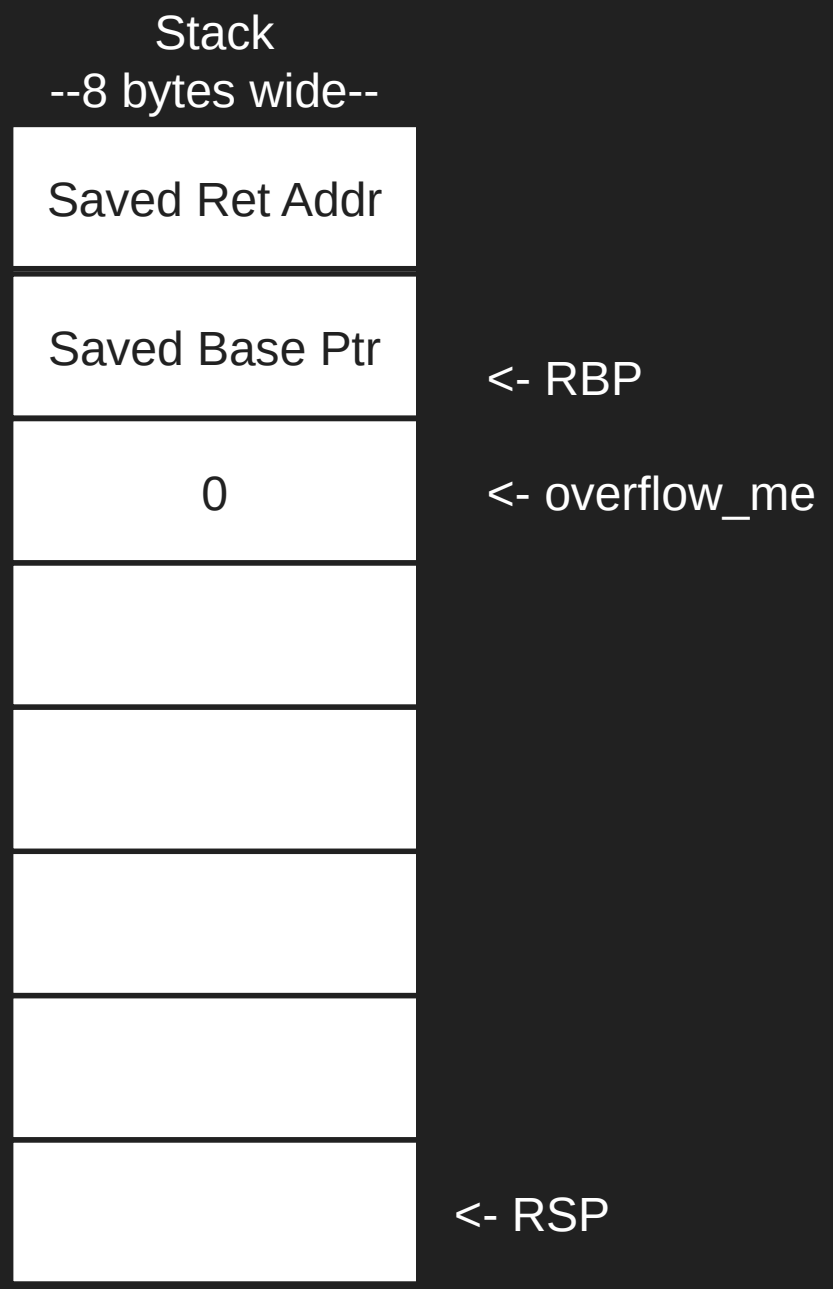


# Corrupting the Stack:

## Lets examine our code

```c
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame
2. Initialize overflow_me



# Corrupting the Stack:

## Lets examine our code

```c
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame
2. Initialize overflow_me
3. Call gets function

# Corrupting the Stack:

## Lets examine our code

```
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame
2. Initialize overflow_me
3. Call gets function
4. Everything looks ok!

# Corrupting the Stack:

## Lets examine our code

```c
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame
2. Initialize overflow_me
3. Call gets function

What if we wanted to give more input?

# Corrupting the Stack:

Lets examine our code

```c
#include <stdio.h>

int main() {

        long overflow_me = 0;
        char buf[32];

        gets(buf);

        printf("%ld\n", overflow_me);

        return 0;
}
```

1. Setup Stack Frame
2. Initialize overflow_me
3. Call gets function

What if we wanted to give more input?


# What if we wanted to give even more input!

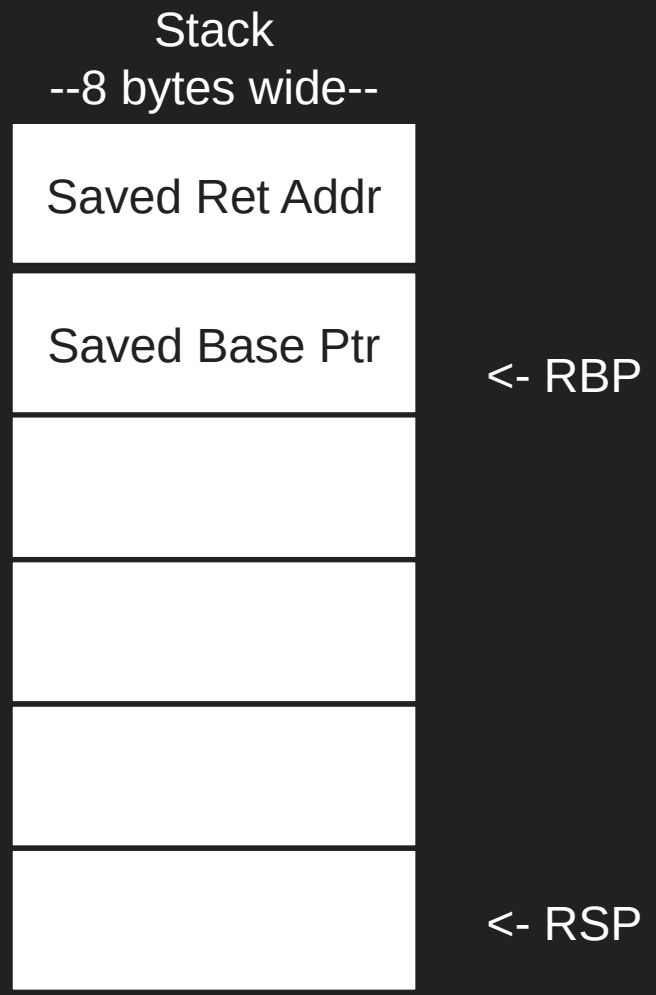
# So we crashed the program…

- Overwriting the return address and base pointer messes up the execution of the rest of the program
  - Base pointer is important because we index local variables with it
  - Return address being corrupted means we'll jump to an invalid location after we're done with main (the program doesn't end after main it still needs to clean up some stuff)

Can we fix this?

# New Program!

```c
void win() {
        printf("You win!\n");
}

int main() {
        char buf[32];

        gets(buf);

        return 0;
}
```

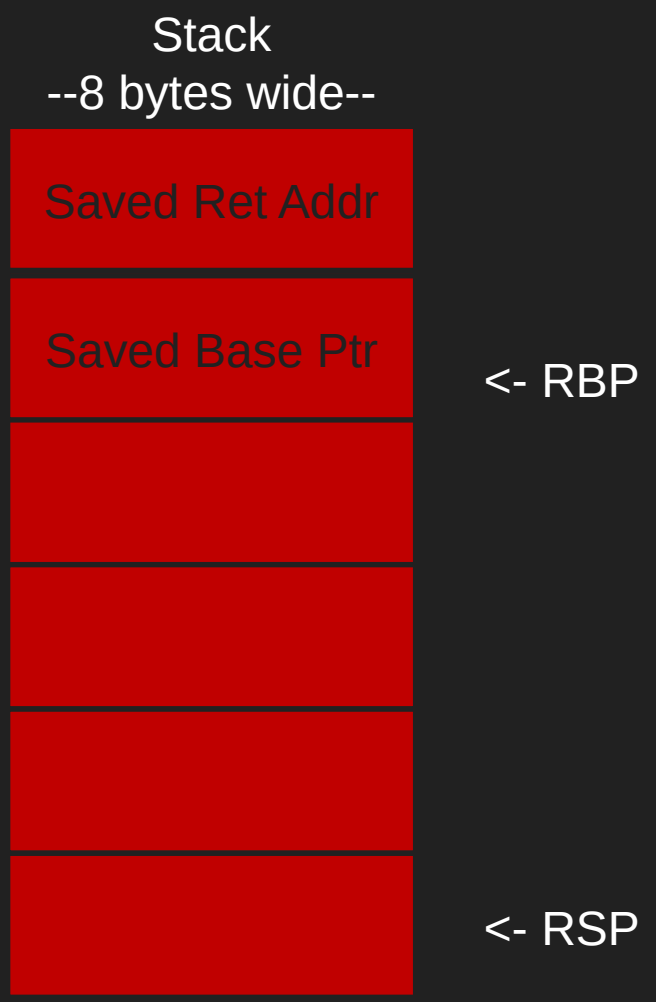


# New Program!

```
void win() {
        printf("You win!\n");
}

int main() {
        char buf[32];

        gets(buf);

        return 0;
}
```

Just like before we can corrupt the memory and saved values and crash the program

How do we get to the win function though?

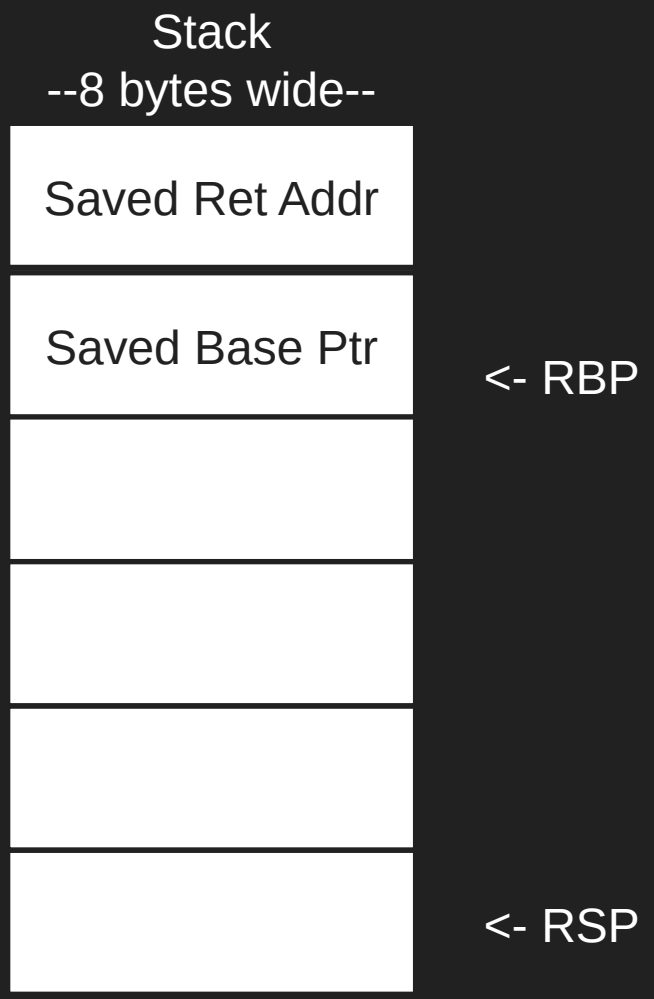


# Lets look into memory:

```c
void win() {
        printf("You win!\n");
}

int main() {
        char buf[32];

        gets(buf);

        return 0;
}
```

Since our programs don't have mitigations on them, they are loaded into predictable memory ranges

If we disassemble main and win in gdb, we can get the location in memory they are put into during every execution



# Our New Exploit:

```c
void win() {
        printf("You win!\n");
}

int main() {
        char buf[32];

        gets(buf);

        return 0;
}
```

1. Fill the stack until we reach the base pointer
2. Overwrite the base pointer (doesn't really matter what the value is)
3. Set return address to our intended destination (win offset)
4. Run program and you win!

# Things to Note:

- You can to anywhere in the program
  - Doesn’t have to be the start of a function, you can jump to the center to avoid certain parts or even jump to data

- Order of variables on the stack matter!
  - If the buffer came first on the stack, since we can only write up, we are unable to change local variables below it