CS 395: Binary Exploitation in Linux

Week 1: Integer Overflows & Basic Buffer Overflows

What is an integer overflow?

An integer overflow occurs when an arithmetic operation creates a value larger (or smaller) than what can be represented by the available bits of its data type.

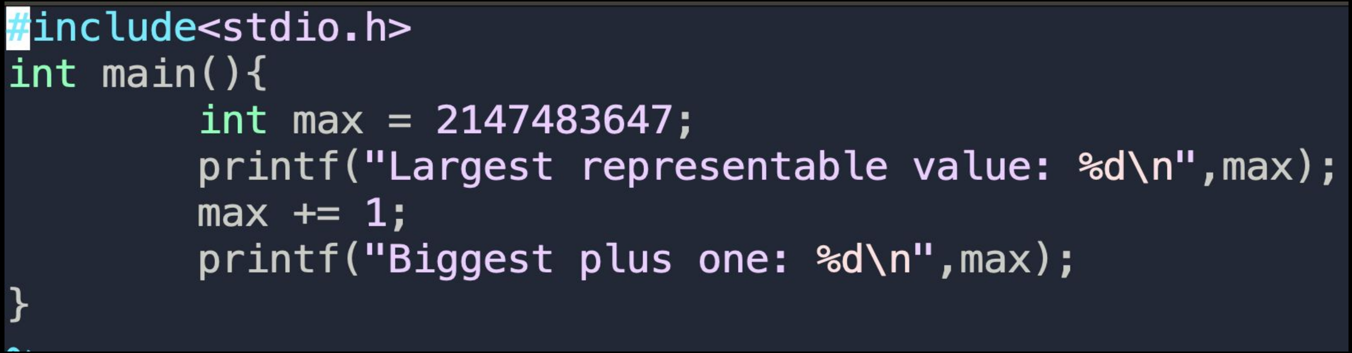
For example, the largest number a standard 32-bit C signed int can represent is:

2,147,483,647 in decimal

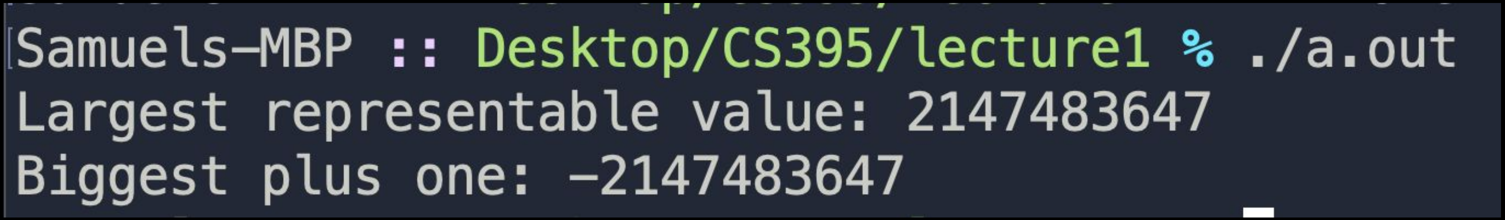
Or 0111111111111111111111111111111 in binary (that’s 31 ones)

So what would happen if we added 1 to this number?

Example



```c
#include<stdio.h>
int main(){
        int max = 2147483647;
        printf("Largest representable value: %d\n",max);
        max += 1;
        printf("Biggest plus one: %d\n",max);
}
```



```
[Samuels-MBP :: Desktop/CS395/lecture1 % ./a.out
Largest representable value: 2147483647
Biggest plus one: -2147483647
```

Why is it negative !?

Explanation

2147483647 decimal = 01111111111111111111111111111111 in binary

01111111111111111111111111111111 + 1 =
2147483647 = 10000000000000000000000000000000
(-2147483648)

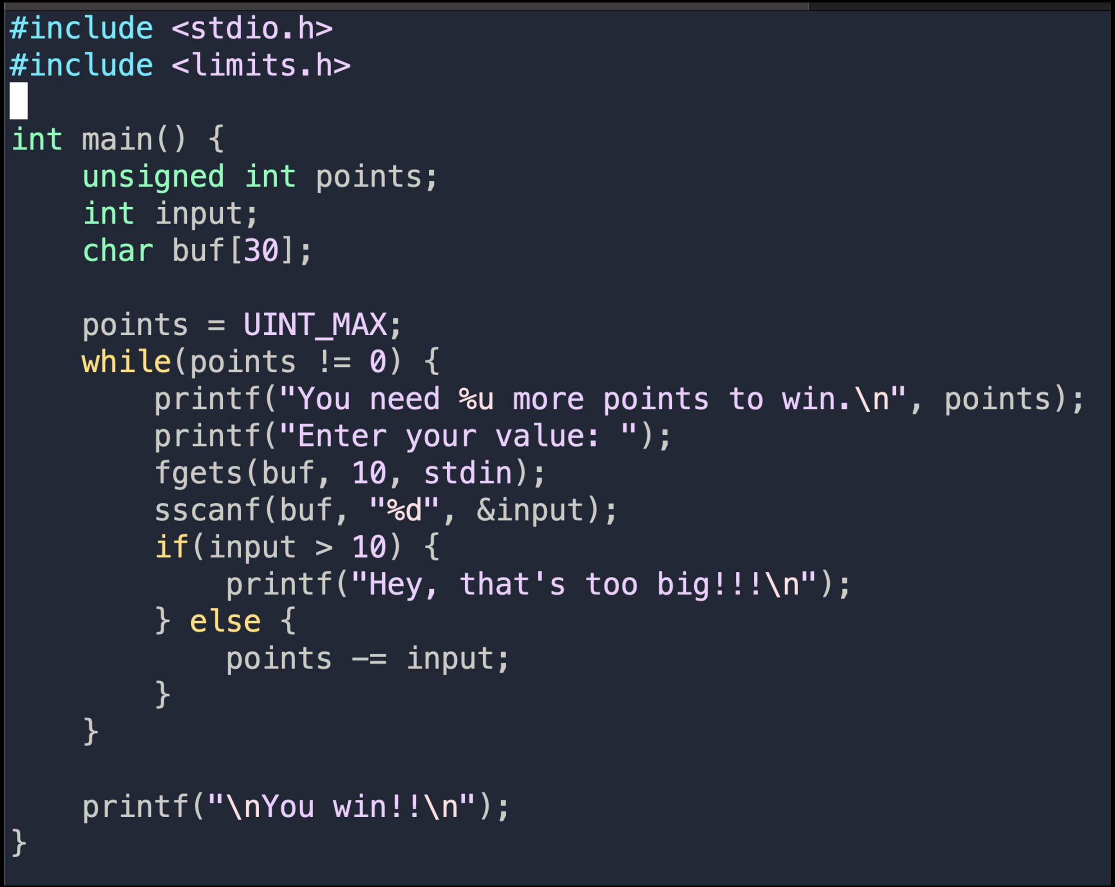
The addition makes the value so big it has to flip the sign bit, which ends up representing a very small number!

If a programmer doesn’t take this possibility into account, someone could take advantage of this mistake and use it to make the program act in unintended ways…

Note: This concept can also be applied to **under**flows, for instance,
-2147483648 - 1 = 2147483647

Note 2: Integer overflows can be especially common when mixing signed and unsigned numbers. Ex: 4294967295 + -1 = -2

Demonstration



```c
#include <stdio.h>
#include <limits.h>

int main() {
    unsigned int points;
    int input;
    char buf[30];

    points = UINT_MAX;
    while(points != 0) {
        printf("You need %u more points to win.\n", points);
        printf("Enter your value: ");
        fgets(buf, 10, stdin);
        sscanf(buf, "%d", &input);
        if(input > 10) {
            printf("Hey, that's too big!!!\n");
        } else {
            points -= input;
        }
    }

    printf("\nYou win!!\n");
}
```

Demo

We need 4294967295 (11111111111111111111111111111111) points to win, but we can’t enter any number bigger than 10!

Our input is recorded as a signed int, but the total points is an unsigned int…

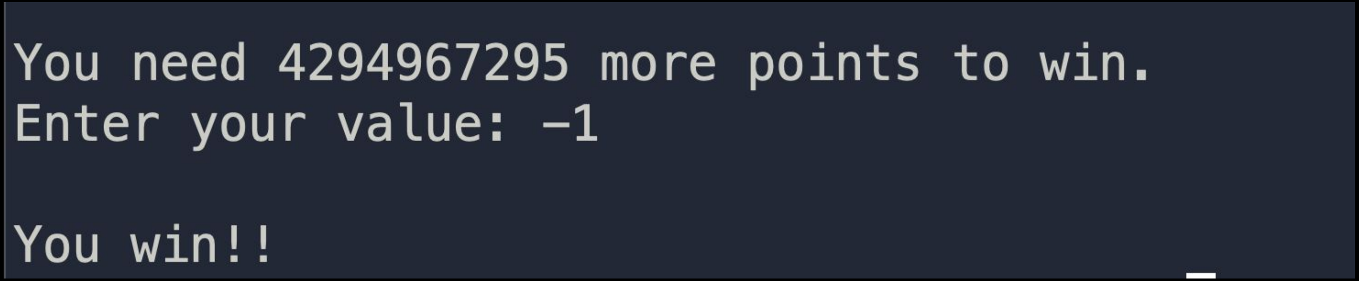
We can use this to our advantage to exploit the program!

Craft an Exploit

4294967295 = 11111111111111111111111111111111

-1 = 11111111111111111111111111111111 and is <10!

4294967295 - -1 = 0, and we win!

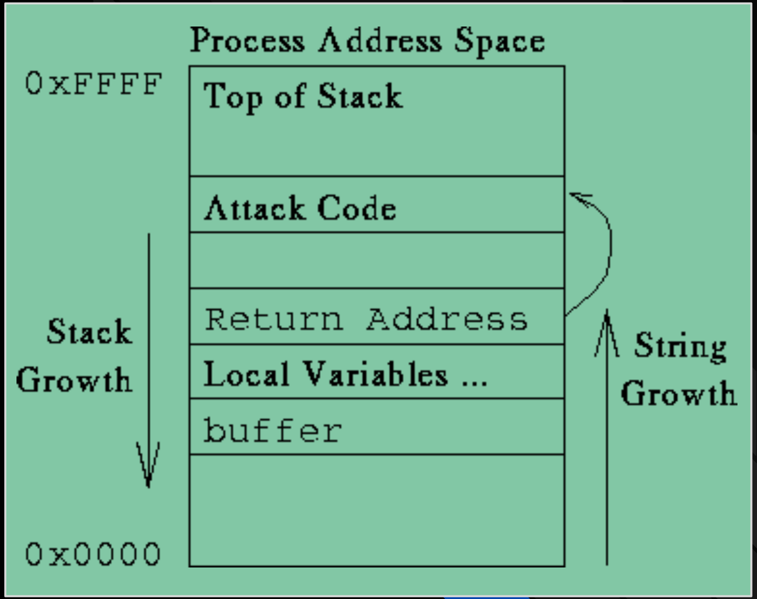


```
You need 4294967295 more points to win.
Enter your value: -1

You win!!
```

Buffer Overflows

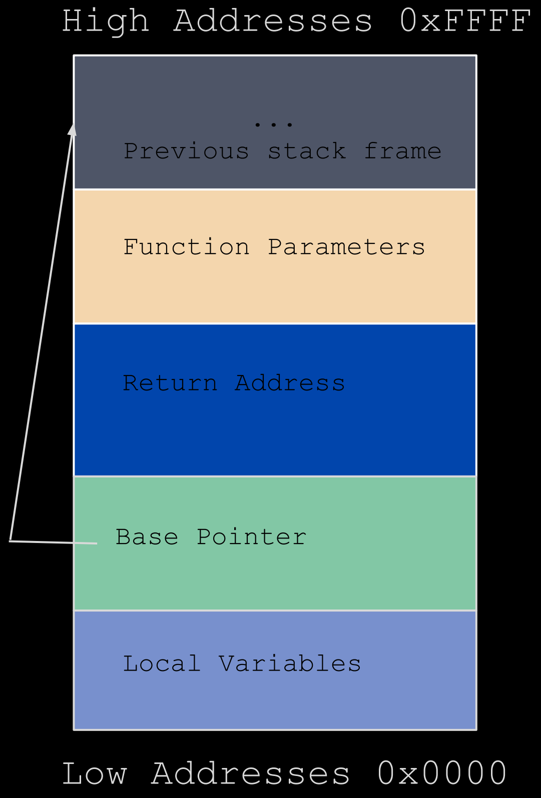
very very important!



https://www.usenix.org/legacy/publications/library/proceedings/sec98/full_papers/cowan/cowan_html/img1.gif

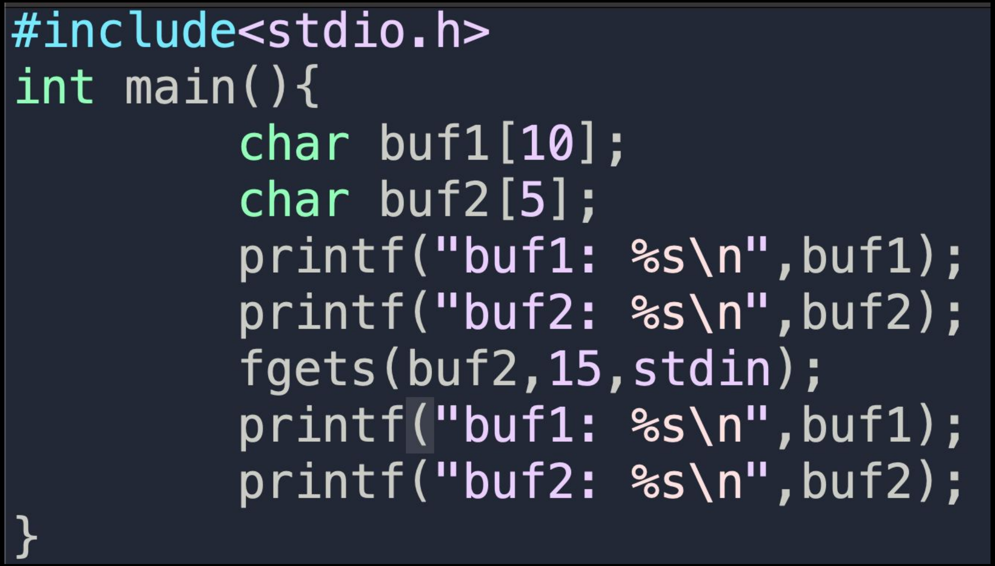
Stack Frames

- Each function gets its own stack frame when it's called at runtime
- Each frame holds its function parameters,return address, base pointer, and local variables.
- The return address points to the code to run after the current function completes
- The base pointer holds the address of the previous stack frame
- The local variables sections holds all the variables local to the function



Overflowing Buffers

C has no array bounds checking built in. This is a big pain when writing secure C code, but offers great opportunities for mischievous hackers. Can you spot the problem with this code?

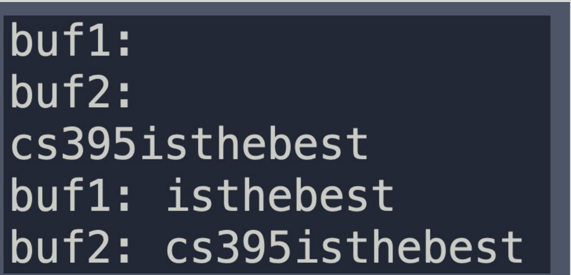


```
#include<stdio.h>
int main(){
        char buf1[10];
        char buf2[5];
        printf("buf1: %s\n",buf1);
        printf("buf2: %s\n",buf2);
        fgets(buf2,15,stdin);
        printf("buf1: %s\n",buf1);
        printf("buf2: %s\n",buf2);
}
```

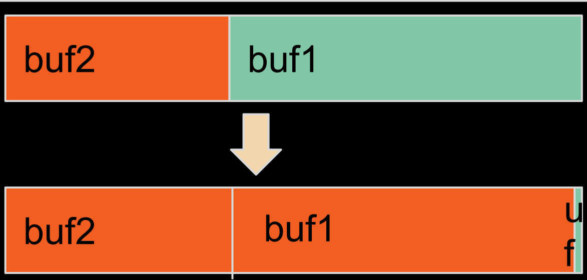
Smashing the Stack

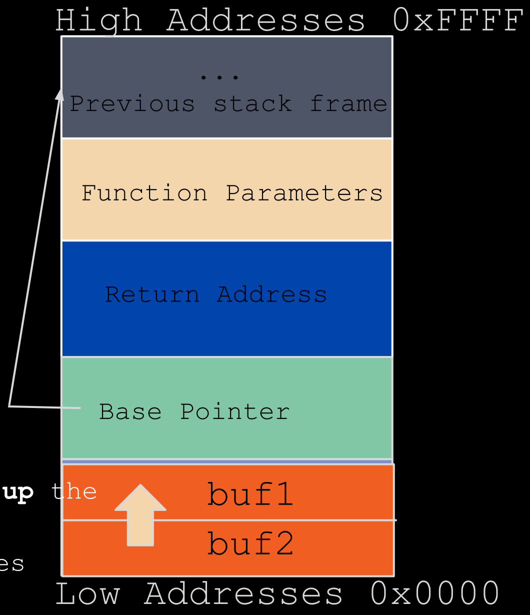
Running the program and entering a string longer than 5 characters causes it to **overflow** from buf2 into buf1, because buf1 is stored above it on the stack.

- The arrays start off empty
- After we enter our input "cs395isthebest" it fills buf2 and overflows into buf1



```
buf1:
buf2:
cs395isthebest
buf1: isthebest
buf2: cs395isthebest
```

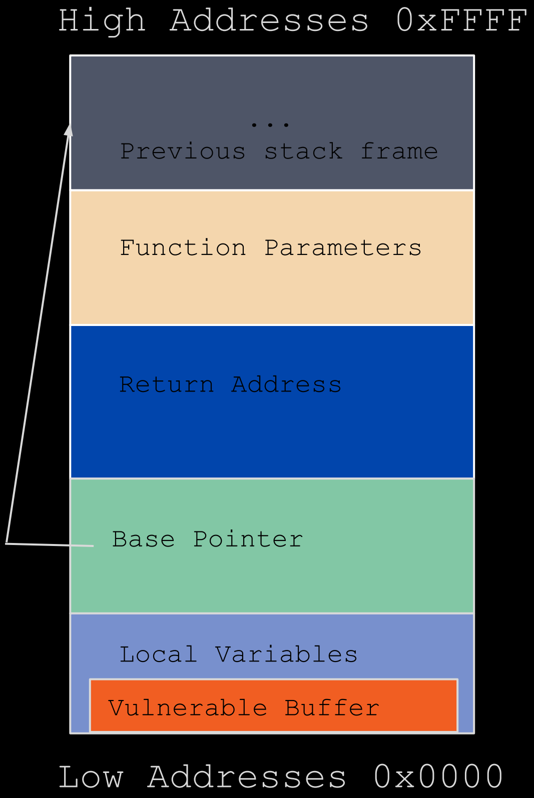




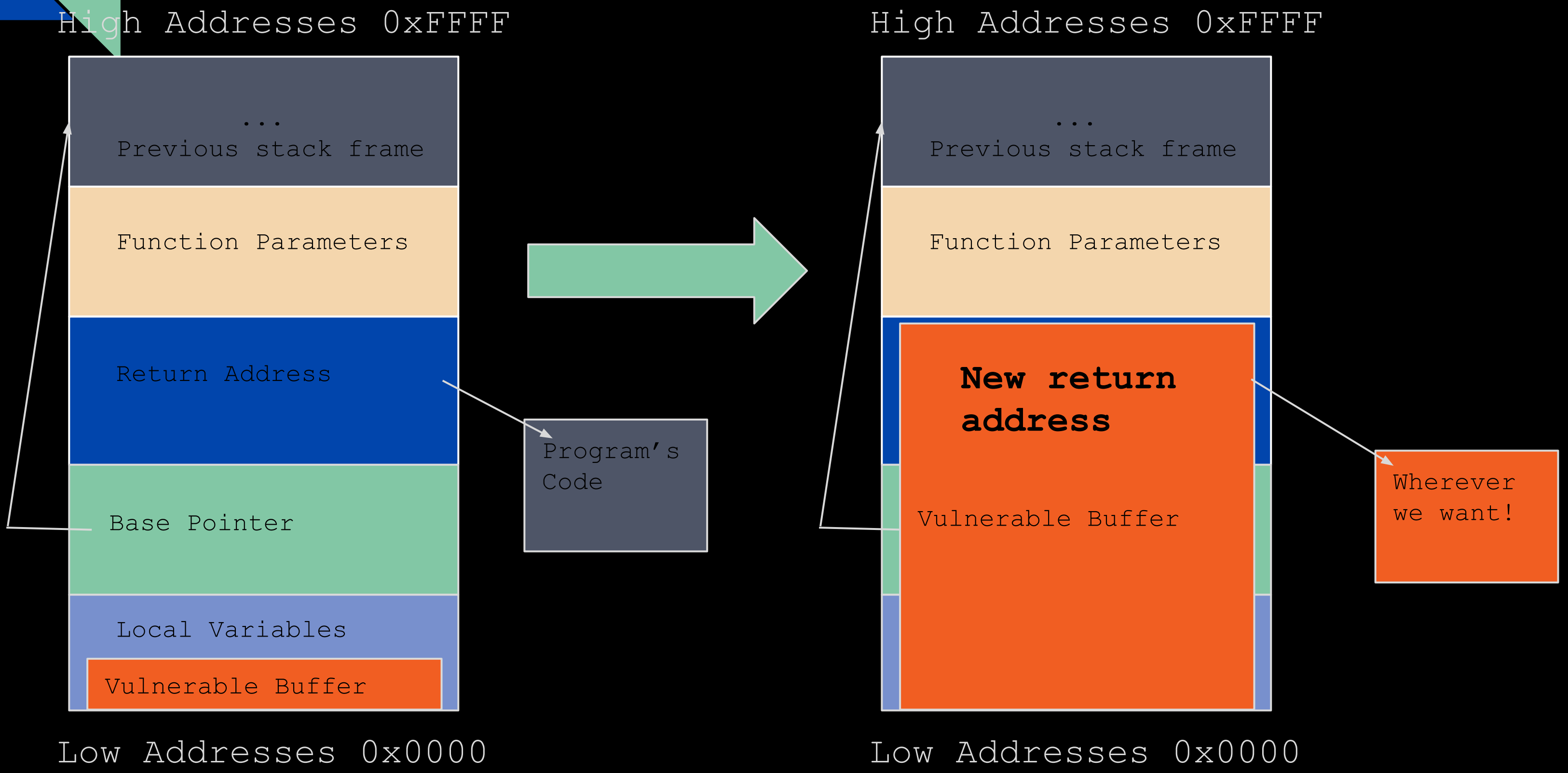
Our input overflows **up** the stack, overwriting everything it touches

Altering Program Flow

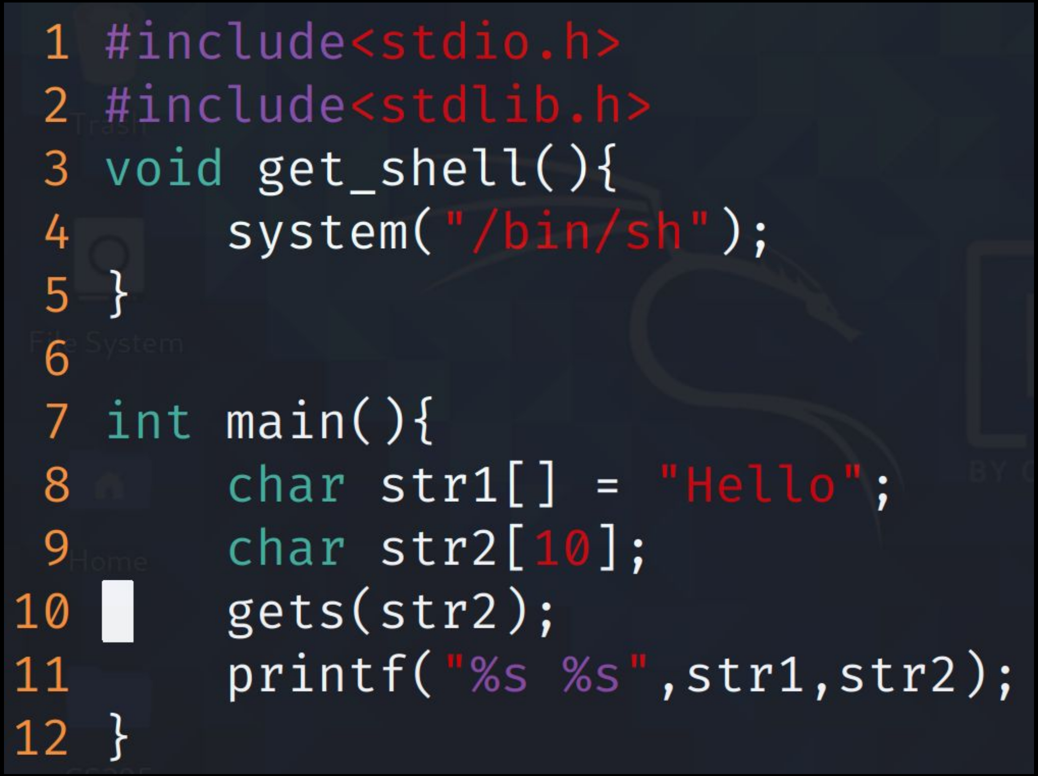
- We know we can replace anything above our vulnerable buffer on the stack.
- How can we use this to 'hack' the program?



The Solution



Demo



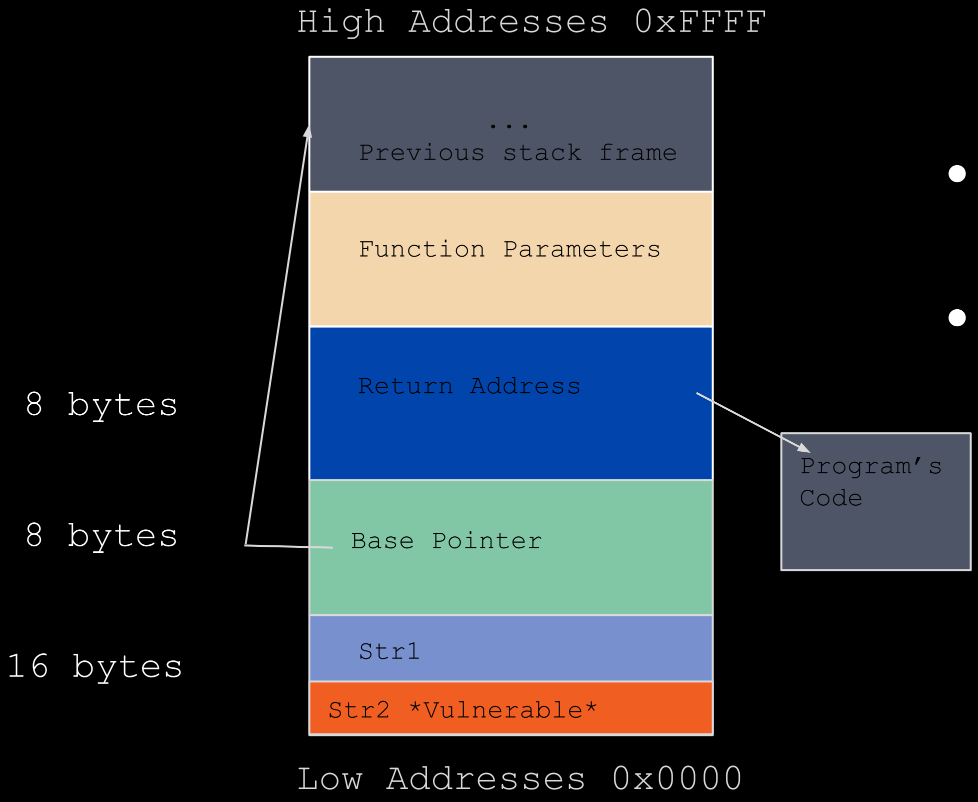
```c
#include<stdio.h>
#include<stdlib.h>
void get_shell(){
    system("/bin/sh");
}

int main(){
    char str1[] = "Hello";
    char str2[10];
    gets(str2);
    printf("%s %s",str1,str2);
}
```

Steps of a Buffer Overflow:

- Find out if our input can crash the program
- Determine the distance from our vulnerable buffer to the return address
- Create a payload: Junk + New return address
- Run program with our payload as input, hack the program.

Creating a Payload



- We want to fill the first 24 bytes up with junk
- then replace the 8 byte return address with our new little endian address

Important Notes:

- **YOU MUST DISABLE ASLR BEFORE TRYING THESE YOURSELF**
    - `./aslr.sh off` <- must be done after every reboot
- Overwriting the Base Pointer on the stack will cause the program to crash after the exploit
- We can’t just type in our exploit through stdin because it is interpreted as ASCII, so we use python as a workaround
- On Intel x64 processors, bytes are ordered in little-endian
- Once our exploit works and we get a shell, to interact with it we must keep stdin open by using a trick like: `(cat payload;cat) | ./vuln`
- Use GDB to see what’s in the registers and watch how your input affects them. GEF helps a lot!

Homework

- There are no writeups due this week.
- Review the basics of programming in Python, C, and assembly on your own.
- Redo the integer overflow and buffer overflow by yourself so that you understand how it works.