CS 395: Binary Exploitation in Linux

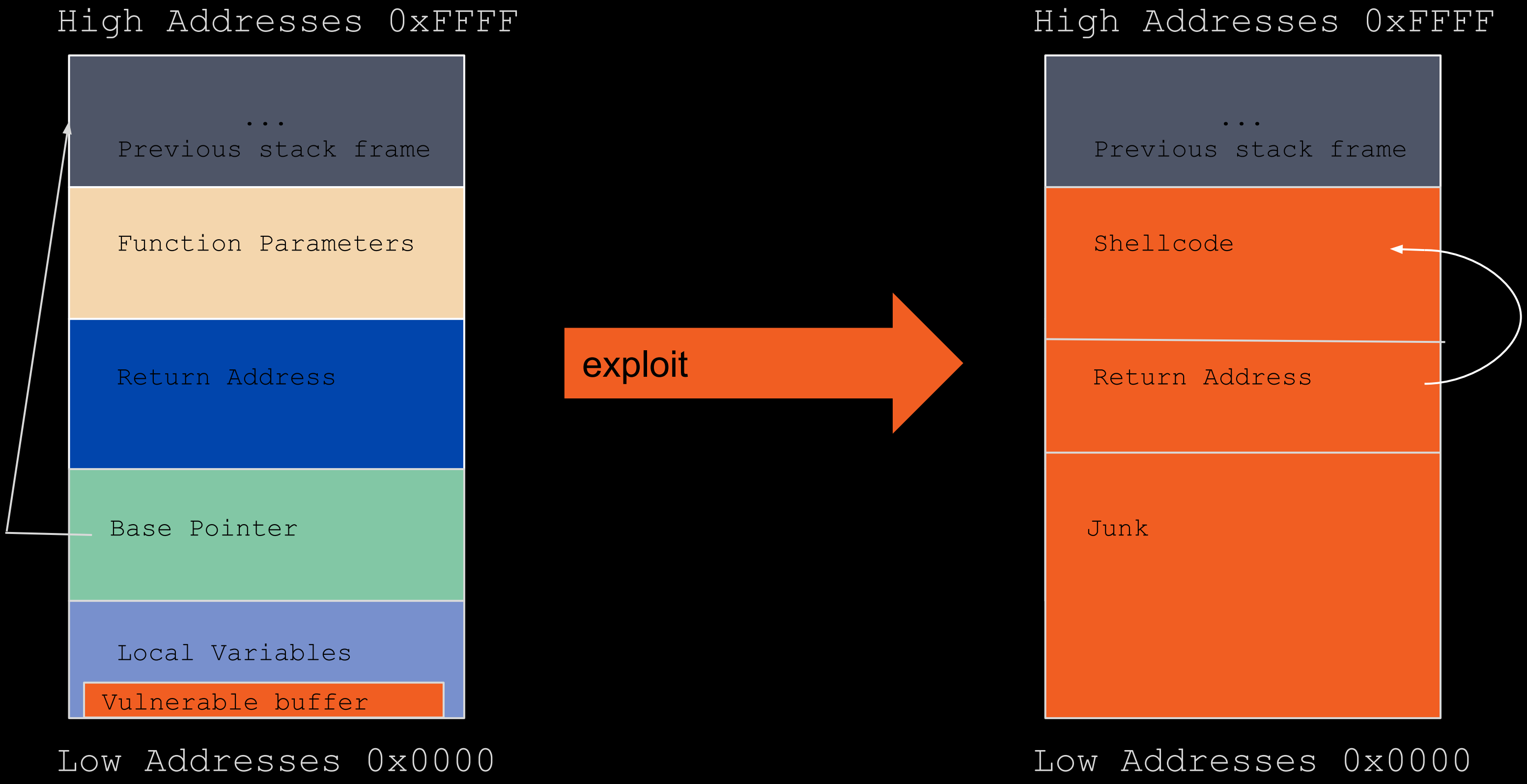
Week 3: Buffer overflows with shellcode and partial overwrites

Why use Shellcode?

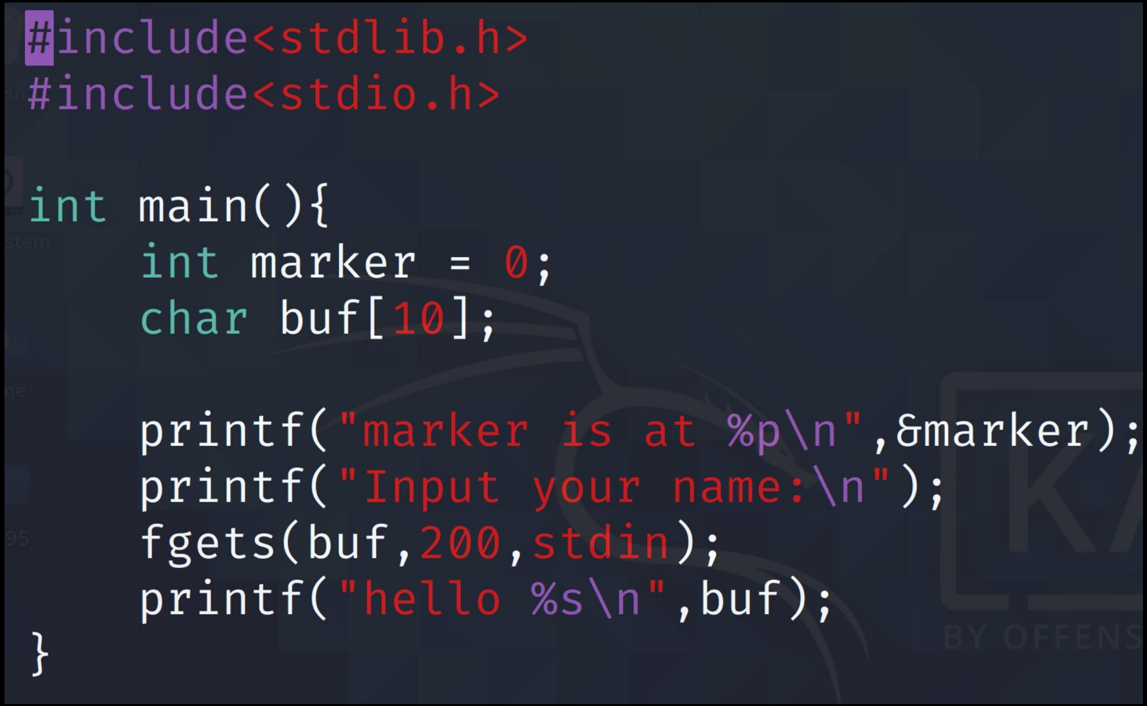
- The vast majority of binaries won’t have a convenient *getshell()* function for us to exploit, so sometimes we have to make our own.
- By using a buffer overflow to push our shellcode onto the stack, we can inject and run whatever assembly we want

Important note: Most programs are compiled with protections against this, and have a non-executable stack. More on this in later lectures.

Pushing our Shellcode



Demo



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

int main(){
    int marker = 0;
    char buf[10];

    printf("marker is at %p\n",&marker);
    printf("Input your name:\n");
    fgets(buf,200,stdin);
    printf("hello %s\n",buf);
}
```

Build a Payload

- The key is to find the address of our shellcode so we can point the return address to it. Luckily, we are given the address of marker, so we can calculate everything we need!

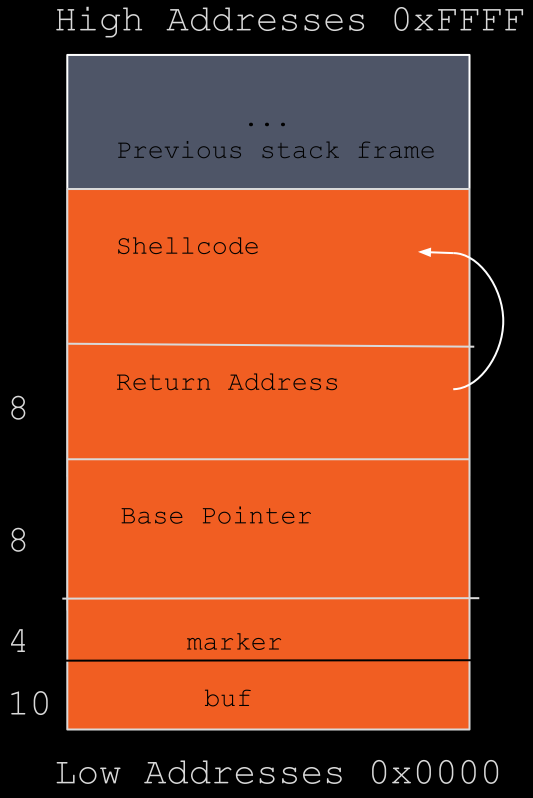
Example:

- If marker is at 0x7fffffffe01c, our return address should be:

  4 + 16 + 0x7fffffffe01c = **0x7fffffffe030**

- So our final payload would be:

(22 bytes of junk) + **0x7fffffffe030** + [shellcode]



High Addresses 0xFFFF

...
Previous stack frame

Shellcode

Return Address

Base Pointer

marker

buf

8

8

4

10

Low Addresses 0x0000

Important Notes

- I solved this challenge with ASLR turned off, but maybe later we’ll learn to do it with ASLR activated....
- The marker address is different when you run it in GDB
- Before we wrote our exploit to a file, now we’re using

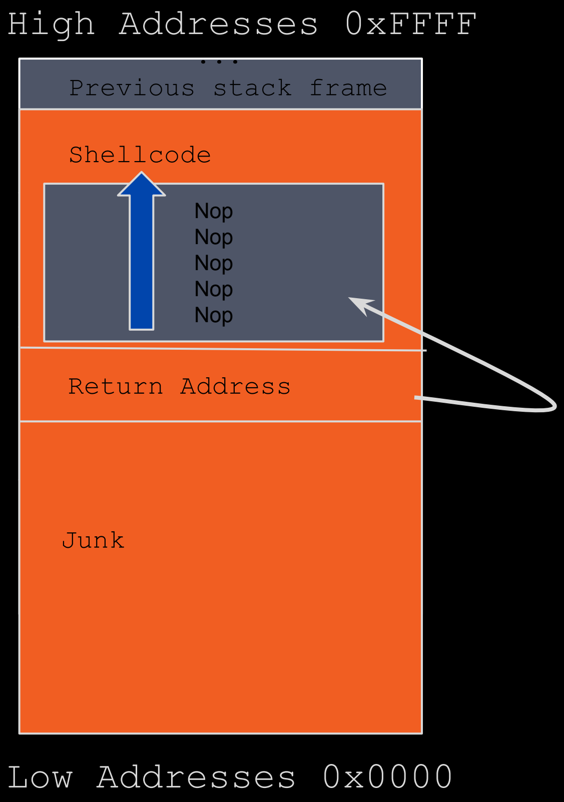
  ```
  python3 -c "import sys;sys.stdout.buffer.write(...)”
  ```
- Either method works, but this one tends to be easier to change quicker
- To use your python script as input in GDB you can use r

  ```
  < <(python3 -c "import sys;sys.stdout.buffer.write())
  ```
- <(command) is bash command substitution in GDB
- To use the script as input from the command line, just use the unix pipe “|” operator

-
  ```
  python3 -c "import sys;sys.stdout.buffer.write()" | ./vuln
  ```

No Exact Marker? Nop a Problem!

- In some situations you don’t have an exact address for your shellcode to replace your return address with
- If you have a rough idea of where your shellcode is and enough room on the stack, you can just pad your shellcode with NOP - the **N**o **OP**eration x64 instruction. (0x90)
- This technique is called creating a **Nopsled**, no matter where your return address points, if it hits a nop it will travel up the stack and eventually hit your shellcode
- We won’t worry about this for now, but in the future it may be helpful in getting around some security...

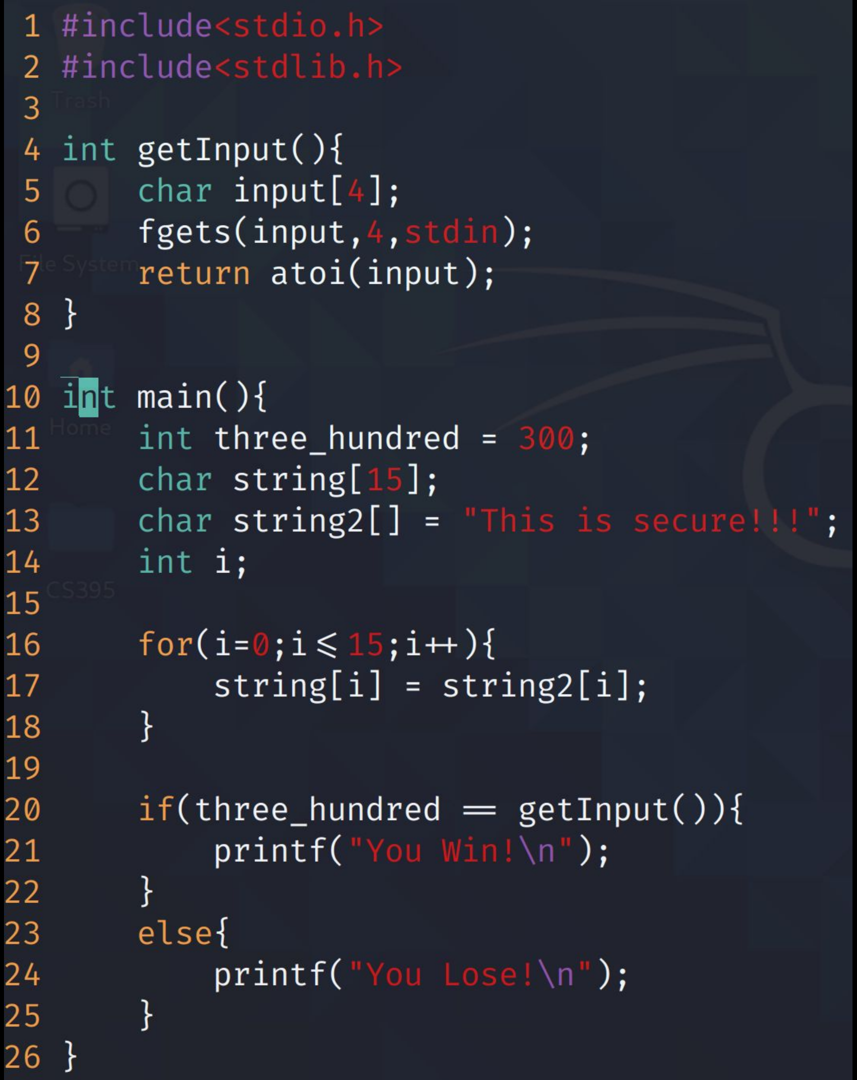


Off by One Overflow

- Sometimes we can exploit a very common array accessing mistake

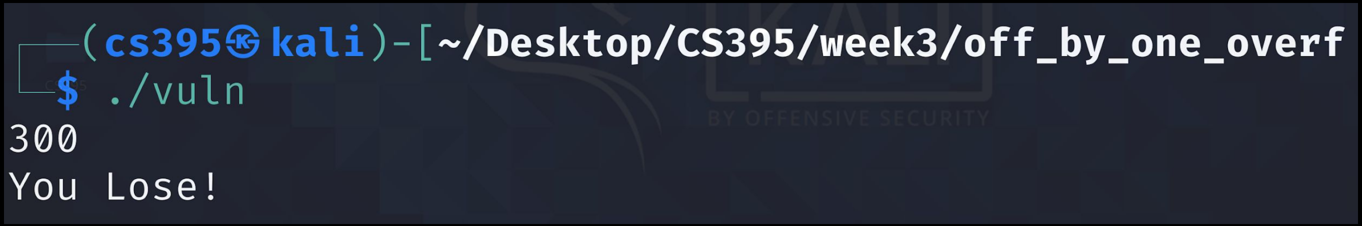
- The for loop that copies string2 to string runs **16** times, when it was meant to only run 15

- Since string[] is directly under three_hundred on the stack, that extra byte overflows into the last byte of three_hundred.



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

int getInput(){
    char input[4];
    fgets(input,4,stdin);
    return atoi(input);
}

int main(){
    int three_hundred = 300;
    char string[15];
    char string2[] = "This is secure!!!";
    int i;

    for(i=0;i<=15;i++){
        string[i] = string2[i];
    }

    if(three_hundred == getInput()){
        printf("You Win!\n");
    }
    else{
        printf("You Lose!\n");
    }
}
```

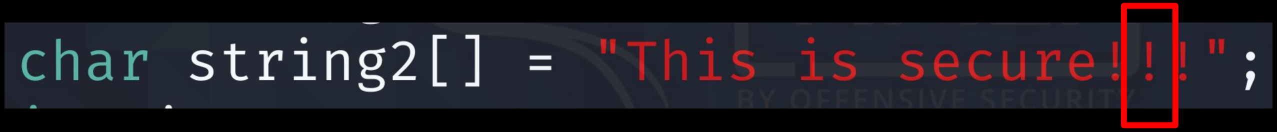
Off by One Overflow cont.



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week3/off_by_one_overf
└─$ ./vuln
300
You Lose!
```

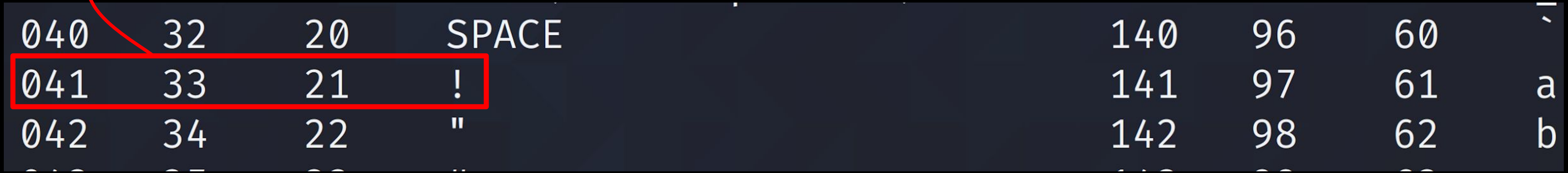
$$300_{10} == 1\underline{00101100}_2$$

This is byte 16



```
char string2[] = "This is secure!!!";
```

! == 33_{10}



```
040    32    20    SPACE                  140    96    60    `
041    33    21    !                      141    97    61    a
042    34    22    "                      142    98    62    b
```

Off by One Overflow Finish

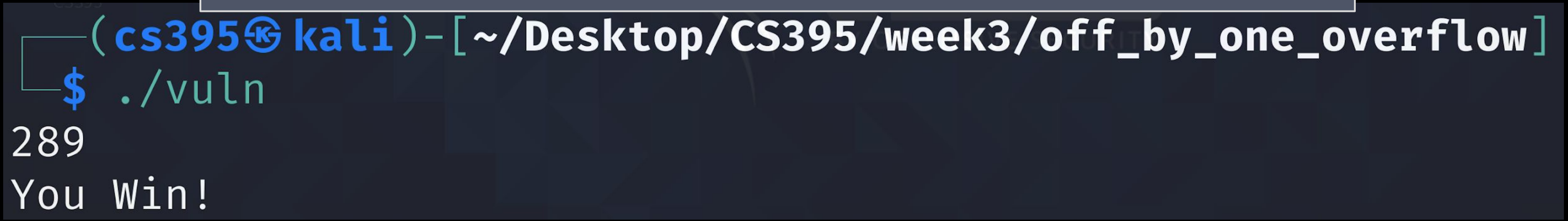
Int three_hundred = 300_{10} == $1\underline{00101100}_2$

After the program runs:

Int three_hundred = 1___!___

! = 33_{10} == 100001_2

Int three_hundred = 100100001_2 == $\mathbf{289}_{10}$



```
┌──(cs395㉿kali)-[~/Desktop/CS395/week3/off_by_one_overflow]
└─$ ./vuln
289
You Win!
```

Homework

Your first assignment is a buffer overflow. You must try to solve it, and submit your solution along with a writeup. The vulnerable binary and its source code are located in ~/Desktop/week3/homework

Things to include in the writeup are:

- Your thought process of exploiting the program, what steps you took and why you tackled the problem the way you did.
- Even if you didn’t succeed, include the things you tried, genuine attempts and creative solutions will be rewarded.
- Be sure to include anything you used in your exploit: payloads, shellcode, techniques, (and later on, scripts)