Reverse Engineering and Binary Exploitation

Laboratory for the class "Security Verification and Testing" (01TYASM/01TYAOV)
Politecnico di Torino – AY 2023/24
Prof. Riccardo Sisto

prepared by:
Cataldo Basile (cataldo.basile@polito.it)

v. 1.3 (9/11/2023)

Contents

Purpose of this laboratory

The purpose of this laboratory is get familiar with reverse engineering tools, which include a static analyser (that is, a disassmbler and a decompiler) and dynamic analysis tool (that is, a debugger and also some tracers, if you prefer).

- First, you will be asked to understand the semantics of simple programs to bypass some controls or force an unexpected behaviour of an application (Section 1).
- Second, you will exploit some programs that are vulnerable to buffer overflows (Section 2).
- Third, you will mount simple *format strings* attacks (Section 3).

Being able to solve all the challenges presented here is not the final purpose of this laboratory. They are just the way to have concrete tasks to do to learn how to use reverse engineering tools and look for the commands you need for some "real" objectives. Therefore, your time is better invested in understanding how to use tools than finding the simplest and fastest way to solve the challenges.

Moreover, we strongly suggest you solve the same challenges using different tools. This approach, especially when you already know what to look for, will ensure you can appreciate the features and advantages of individual tools.

Finally, you will be asked to attack some binaries using ROP, ret2win, and ret2lib. Since these are more complex attacks and may require more time to be understood and executed, you are asked to optionally execute these attacks at your home.

Prepare the environment

For this laboratory, you may use any reverse engineering tool of your choice. You have seen during class Radare2, Ghidra, and gdb.

Radare2 and gdb are available in the VM. Ghidra can be downloaded from

https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_10.4_build/ghidra_10.4_PUBLIC_20230928.zip

If you use the zip file you also have to ensure that Java JDK 17 or JDK 18 are installed on your system and environment variables are properly set (Ghidra did not work well with JDK 21 and JDK 22 in our tests).

You may also have to install Ghidra with

```
sudo apt update
sudo apt install ghidra
```

Execute the tool with the ghidra command or enter the Ghidra installation path and run the script ghidraRun.sh.

You may be interested in improving gdb with some useful scripts for visualization and analysis purposes. You can choose from:

- pwndbg, available here https://github.com/pwndbg/pwndbg;
- peda, available here https://github.com/longld/peda#installation;
- GEF, available here https://github.com/hugsy/gef

All the above links also report installation instructions.

Even if we proposed some tools, feel free to experiment at your place with IDA Pro Free (unfortunately several limitations apply that may prevent you from doing sophisticated tasks, even if the last released free version is quite powerful).

You may also want to try Binary Ninja Cloud, available here (registration needed):

https://cloud.binary.ninja/

Material

This laboratory makes use of source and binary files that have been provided as an archive

lab03_material.zip

If you experience problems in executing some files, it may have happened that permissions have been lost when zipping the material. Make it executable again with the "chmod 777 *filename*" command.

Some hints for automating attacks

When you have to build payloads, you can use Python and send it through the standard input by means of the following pipe:

```
python -c 'print("something to pass via stdin")' | ./prog_to_crack
```

Analogously, you can use Python to build the command line arguments:

```
./prog_to_crack $(python3 -c 'print("argument")')
```

In alternative, even if it requires some initial learning effort, it is very convenient to learn how to use the pwntools, a Python library that automates several attacking operations.

You can find the official documentation here:

http://docs.pwntools.com/en/latest/

and a tutorial here:

https://github.com/Gallopsled/pwntools-tutorial#readme

Finally, you can forge payload with Python also when you use `gdb` to dynamically execute a program, both via standard in:

```
run <<< $(python3 -c 'print("something to pass via stdin")')
```

or

```
run <<< $(python3 myscript.py')
```

and to forge arguments:

```
run $(python3 -c 'print("cmdline")')
```

Below, we have also reported some references to x86 opcodes that you may want to read to refresh your knowledge of the ASM:

http://ref.x86asm.net/coder32.html
https://pnx.tf/files/x86_opcode_structure_and_instruction_overview.png
https://pwn.college/

1 Crack-me challenges

This section includes three exercises that will require you to perform static and dynamic analysis to understand the behaviour of a target program and tamper with it.

Exercise 1

Purpose:	The purpose is to force the program to print the string ”This is the answer!” by patching the binary. You have to force the execution to reach the proper branch, it is not just needed to print the string above.
Suggested tool:	Radare2.
Alternative tools:	Ghidra. You can also use `gdb` to follow the program's execution (step-by-step) and understand where to intervene.
Other proprietary tools and their free versions:	IDA Pro, Binary Ninja.
Hints:	look for comparisons and bypass the correct one.

As a first exercise, you have to crack yourself the binary of the example shown during class. The binary file is available in the lab03 material file

```
/lab03_material/crackme/crackme1
```

Make a copy of this file, as you will have to patch the binaries. You should not access the source code (the binaries have not been stripped down to allow you to reconstruct at least the function names and some useful symbols). However, you can find the source code in the material if you get initially lost with the assembly code.

NOTE

Here is a sequence of commands for `radare2` in case you would fill lost after opening it with only square brackets and numbers...

- `aaaa`, full static analysis of the binaries, feasible and fast as the programs of this lab are very small
- `afl`, get the list of all the functions;
- `pdf`, disassembles and prints the function where `radare2` is currently pointing (i.e., where you performed the last seek);
- `pdf function_name`, disassemble the function whose name is passed as option
- `pd 1`, disassemble the first instructions starting from the current position
- `wx opcode`, write from the current position the opcode passed as a hex string.

You can find more data on the slides and the linked cheatsheets.

ATTENTION

Patching with `Ghidra` is not as “natural” as with `radare2`, this is one of the reasons to start with `radare2`. Nonetheless, if you want to use `radare2` to patch binaries, these need to be opened with the `-w` options.
In `radare2` the `wx` option is the preferred one to patch binaries, read `https://r2wiki.readthedocs.io/en/latest/options/w/wa-push/` for further references.

Exercise 2

Purpose:	find the user's password and bypass the authentication procedure. You can avoid patching the binaries.
Suggested tools:	Ghidra (decompiler), gdb (debug the application), Radare2.
Hints:	execute the program and try to guess its behaviour. Make a mental representation of the program. Disassemble and find what to bypass (using your guesses on the program structure).
Alternative solution:	look for hardcoded values and find what to do with them.
Source:	Root me

This is another simple crack taken from the Root Me challenge repository. The binary file is available in the lab03 material file

```
/lab03_material/crackme/crackme2
```

Exercise 3

Purpose:	find the key (it is an integer value).
Suggested tools:	Radare2, Ghidra, gdb.
Hints:	execute the program and try to guess its behaviour. Disassemble and find where the key comparison is performed then discover the value. Use Radare2 first, if the task look too hard, switch to Ghidra
Source:	Crackes.me

NOTE

Due to a mistake in the scanf management, this program is bugged. If your input does not start with a digit, it already returns the success string. For this exercise, you have to input digits if you want to seriously exploit the application.

This is another crack-me file taken from the Crackes.me challenge repository. The binary file is available in the lab03 material file

```
/lab03_material/crackme/crackme3
```

Did you appreciate the role of the decompiler and why is it worth paying a lot for it?

2 Exploiting buffer overflows

This section includes five exercises to learn how to exploit buffer overflow weaknesses in the code. You will have to perform static and dynamic analysis to find the information you will need to execute the correct exploits.

Exercise 1

Purpose:	you have to bypass the constraint in the `func` function and make the program print the string “This is the answer!” by accessing the proper program branch (it is not just needed to print the string above).
Suggested tools:	pwntools, shell + python, gdb.
Hints:	Find the payload that helps bypass the condition.

As a first exercise, you have to exploit a buffer overflow in the example application shown during class. The binary file is available in the lab03 material file:

```
/lab03_material/crackme/crackme1
```

You also have access to the source code.

Build the payload *directly on the shell*, with a Python in-line command, then use a pipe to pass it to the target program

Exercise 2

Build the exploit described in Exercise 1 in (at least) two additional methods:

- *using `gdb`*, you should already know the payload. Put a breakpoint in the main, then disassemble the `func` function and add another breakpoint before the instruction calling the input. Then, use the `gdb` visualization options to inspect the stack before and after the input is collected with this command:

  ```
  x/192x $esp
  ```

- *with the `pwntools`*, use the command/object `process` to execute an application, build the payload by concatenating the strings you want to pass, then use the `sendline` command to pass your payload to the process.

Exercise 3

Purpose:	bypass the condition and get the string “You won!”.
Suggested tool:	pwntools, shell + echo, shell + Python (may not work on ZSH), gdb.
Hints:	Pay attention to the endianness. Consider using `echo -ne` to bypass shell input manipulation.
Author:	AS (pwnthem0le)

This is another classic buffer overflow exercise. The binary file is available in the lab03 material file:

```
/lab03_material/bof/bof3
```

The source code is also available in the material file:

```
/lab03_material/bof/bof3.c
```

Exercise 4

Purpose:	bypass the condition.
Suggested tool:	gdb, then pwntools, shell + echo, shell + Python (may not work on ZSH).
Hints:	find the address of the function by dynamically executing the program.
Author:	AS (pwnthem0le)

This exercise will ask you to work with function pointers. Again, this is a classic buffer overflow exploit. The binary file is available in the lab03 material file:

`/lab03_material/bof/bof4`

The source code is also available in the material file:

`/lab03_material/bof/bof4.c`

Exercise 5

Purpose:	reverse the binaries then exploit the buffer overflow to bypass the constraint. Do not patch the binaries.
Suggested tool:	Radare2, Ghidra, then pwntools, shell + python.
Hints:	Find the important function, then disassemble or decompile the program. Plan the payload to inject to successfully mount the exploit.
Source:	cracksme.one

This exercise will ask you to perform some reverse engineering before understanding how to exploit a buffer overflow.

The binary file is available in the lab03 material file:

`/lab03_material/bof/bof5`

Exercise 6

Purpose:	spawn a shell by exploiting the buffer overflow.
Suggested tool:	pwntools, shell + python, gdb.
Hints:	Find the payload that helps bypass the condition.

ATTENTION

If you experience problems and crashes when the shell is spawned with the 32-bit application, use the 64-bit no-pie version (PIE = Position Independent Execution). Nonetheless, the important part of this lab is to understand how to mount and what bytes you have to send by using reverse engineering tools. If you are spending too much time on it when you are in the lab, you better ask me if the approach is correct, jump to the next exercises and do yourself at home the rest of the fine-tuning...

Using the same binary file as Exercise 1, try to spawn a shell. On your machine, it will not be a great success (as you already own that machine) but consider that these attacks are intended to be executed on remote servers. The binary file is available in the lab03 material file:

`/lab03_material/crackme/crackme1`

You can access the source code:

```
/lab03_material/crackme/crackme1.c
```

HINT: you can find the shellcode on shellstorm http://shell-storm.org/shellcode/ (look for the proper platform) or you can generate directly using the `shellcraft` module in a python program).

```
from pwn import *

#...

context.update(arch='i386', os='linux')
shellcode = shellcraft.sh()
payload = asm(shellcode)
```

ATTENTION

In some cases, the addresses that you in `gdb` are different from the actual ones when the program is executed from the shell. Consider some minor shifts (some bytes) when mounting the attack. So, either you execute scripts on the same environment where you collect addresses or add NOPS and hope they are enough.

```
#include <stdio.h>

int main()
{
       int a=1,b=2,c=3;
       int n1,n2;

       printf("%d %d %d\n",a,b,c);
       printf("%d %d \n",a,b,c);
       printf("%d %d %d\n",a,b);
       printf("%2$d %3$d %1$d\n",a,b,c);

       printf("The printf can store the number of bytes written to stdout up to
          this point%n (this number is stored in n1) and the ones up to this
          point%n (this number is stored in n2)\n", &n1, &n2);
}
```

Figure 1: A simple `printf` example (/lab04_material/fs/prinft_example.c).

3 Format strings: a dangerous type of injection

This section provides exercises asking you to perform format string attacks. Some details on the attack technique are provided first.

The `printf` function

Let's start this set of exercises with a simple example presented in Figure 1.

You are certainly familiar with the `printf` functions, but let's repeat some basic concepts as the perspective of software programmer and attacker are different.

Normal printing. Starting from this instruction, argument how the format strings (\%d) are associated to the variables.

```
printf("%d %d %d", a,b,c);
```

It is important to understand: What actually are the symbols `a,b,c`? Where are they stored? How are the symbols `a,b,c` used in order to retrieve the values that are actually printed?

→

More variables. Now, discuss what happens when there are more variables than format strings, like in the instruction below:

```
printf("%d %d", a,b,c);
```

→

More format strings. On the contrary, what happens if there are more format strings than variables from which to take the values?

```
printf("%d %d %d", a,b);
```

```
include <stdio.h>

int main(void) {
       char buffer[128];

       printf("Insert a string: ");
       gets(buffer, sizeof(buffer), stdin);
       printf(buffer);

       return 0;
}
```

Figure 2: Typical weakness that allow format string attacks (lab04_material/fs/memory_read.c).

What is the value printed on the screen? Discuss your opinion on where the `printf` has taken the last printed value.

→

Direct access to variables. Moreover, the `printf` allows us to directly access the passed arguments:

```
printf("%2$d %3$d %1$d", a,b,c);
```

Can you comment on what is the code above performing? (write a dummy program and execute it)

→

Can you figure out the mechanism the `printf` uses to number and refer to the parameters it accesses?

→

Write printed characters. Finally, the `printf` can also save data in memory, as in the following instruction:

```
printf("The printf can store the number of bytes written to stdout up to this
    point%n (this number is stored in n1) and the ones up to this point%n (this
    number is stored in n2)\n", &n1, &n2);
```

The purpose of the `\%n` `printf` feature was introduced to manage well-formatted and aligned ASCII-based GUIs.

NOTE

It is important to remember that `&n1` is the address where to write the counter of the printed characters.

Reading the memory

Let us consider the program presented in Figure 2. In this case, you can control what to pass to the `printf` function. What is the most useful payload you can pass?

→

Take advantage of the Python scripting to pass a format string that allow you to print the content of the stack.

→

Try now with a marker, like with the following in the command line below:

```
python -c 'print("AAAA"+".%x"*10)' | ./fs1
```

to determine where the "AAAA" is in the stack (i.e. how many 4-byte blocks from the beginning of what has been pointed out by the printf function).

→

Use direct access to the parameters to only print the marker twice.

Can you imagine how to use the approach with markers just presented to read arbitrary values in memory? What should you use instead of the "AAAA" to read the content of a specific address?

→

Try now to apply the concepts seen here to solving the following exercises.

Exercise 1

ATTENTION

Before starting these exercises, you should disable the ASLR, otherwise all the segments will be allocated in random memory positions. This is possible by running this command:

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

The ASLR will be disabled until the next restart.
Should you need to enable it again before the restart you can run

```
echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
```

Purpose:	apply a format string attack to read the content of a local variable.
Suggested tool:	pwntools, shell + python.
Hints:	Find the memory location address to read then use the examples seen before to read the content (the marker example). Observe the stack, compare its address with the address of the PIN, and then reason about where the buffer is to make a simpler attack.

Find a way to print the value of the PIN variable by exploiting the format strings.

The binary file is available in the lab04 material file:

```
/lab04_material/fs/fs1
```

The source code is also available in the material file:

```
/lab04_material/fs/fs1.c
```

NOTE

You can get the address of variables in the stack using gdb. Add a breakpoint (e.g. to the main function), then disassemble the main and add another breakpoint just before the execution of the format string injection. You can inspect the whole stack with

```
x/nx $esp
```

where *n* is the number of bytes in the stack you want printed.
In any case, the application you have to tamper with prints out the address of the important variables. Therefore, just look at the output. You will also notice that the addresses differ when you execute the application normally or debug it.

Exercise 2

Purpose:	apply a format string attack to read the content of a global variable.
Suggested tool:	pwntools, shell + python.
Hints:	Find the memory location address to read then use the examples seen before to read the content (the marker example).

Find a way to print the content of the password, saved as a global, which has been declared as a variable.

The binary file is available in the lab04 material file:

```
/lab04_material/fs/fs2
```

The source code is also available in the material file:

```
/lab04_material/fs/fs2.c
```

NOTE

You can get the address of a variable in the string by means of gdb. Add a breakpoint (e.g. to the main function), then use (assuming the symbols have not been stripped down):

```
x/x &variable_name
```

to print the address of the variable, in this case, the pointer to the beginning of the string. This is possible if ASLR is not enabled, as in this case. Otherwise, use rabin2 to check first.

4 Additional challenges available on the web

This repository contains several crack-me challenges (aka *rev*), nonetheless, the distinction between challenges that only require reversing and the ones that require tampering (*pwn* challenges) is not always that evident

https://crackmes.one/search

Root Me also contains a section dedicated to crack-me challenges, they range from basic to medium-level ones.

https://www.root-me.org/en/Challenges/Cracking/

WeChalls is another website that contains interesting challenges in increasing order of difficulty:

https://w3challs.com/challenges/list/reversing
https://w3challs.com/challenges/list/pwn

This site, which also has additional “wargames”, proposes Narnia to learn basic exploitation:

https://overthewire.org/wargames/narnia/

This tutorial also shows how to use format string attacks to overwrite the GOT and tamper with applications:

https://axcheron.github.io/exploit-101-format-strings/

If you want to find more advanced (pwn) challenges (and very well-done tutorials)

https://pwnable.kr/

This website introduces to ROP attacks:

https://ropemporium.com/

Concerning the format string attacks, you can learn how to write values in memory with the `\%n` by reading these tutorials:

https://secgroup.dais.unive.it/teaching/security-course/format-strings/
http://marcin.owsiany.pl/sec/format-string-attacks.pdf

A little more advanced attacks

5 Return Oriented Programming

This section will show you some examples of Return Oriented Programming. This technique becomes necessary when the NX (no-execute) flag is set, thus a simple Buffer Overflow would not work. You will have to perform static and dynamic analysis to solve the following three exercises.
These attacks are much more complex than the previous ones. It is not expected that you are able to solve them during the lab. During the next days we will publish hints on how to solve them, if you didn't succeed by yourself.

Exercise 1

Purpose:	modify the program control flow in order to make the program print the content of `flag.txt`.
Suggested tools:	pwntools, shell + python, gdb, Ghidra.
Hints:	The program already implements a function that executes "cat flag.txt"
Source:	ROP Emporium

As a first exercise, you have to use ROP in a version usually called ret2win, since the program will return to a "win function". The binary file is available in the lab03 material file:
`/lab03_material/rop/rop1`
As for the buffer overflow exercises, it is suggested to build the exploit in (at least) three different methods:

- *directly on the shell*, build the payload with a Python in-line command, then use a pipe to pass it to the target program
- *using* `gdb`, you should already know the payload. Put a breakpoint in the main, then disassemble the `pwnme` function and add another breakpoint before the instruction where the input is collected. Then, use the `gdb` visualization options to inspect the stack before and after the input is collected. `x/192x $esp`
- *with the* `pwntools`, use the process `process` to execute an application, build the payload by concatenating the strings you want to pass, then use the `sendline` command to pass your payload to the process.

Hints:

- in x86 processors the calling convention does not use the stack to pass the parameters, instead the registers are used. The first argument goes into the RDI register, the second one in RSI, etc. More details in this interesting page:
https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/linux-x64-calling-convention-stack-frame
- first force the application to crash and determine how many bytes you need to send to overwrite the return address (you may want to investigate how useful the pwntools function `cyclic` is in these cases);
- find the address of the ret2win function (any static analysis tool will work fine);
- you may have to add some address of a RET in the payload (see https://ropemporium.com/guide.html, "Common Pitfalls", "The MOVAPS issue").

ATTENTION

Some versions of GLIBC use the `movaps` instructions to move data onto the stack in certain functions. The 64 bit calling convention requires the stack to be 16-byte aligned before a `call` instruction but this is easily violated during ROP execution. Make sure to align the stack to 16 bytes before calling e.g. a `system()`.

Exercise 2

Purpose:	modify the program control flow in order to make the program print the content of `flag.txt`.
Suggested tool:	pwntools, shell + python, gdb, Ghidra.
Hints:	Find a way to modify the parameter of an already present `system()`.
Source:	ROP Emporium

This challenge requires more steps than the previous one and it is for sure more challenging. You have to build a so called ROP chain. The binary file is available in the lab03 material file:
`/lab03_material/rop/rop2`

Exercise 3

Purpose:	spawn a shell exploiting ROP.
Suggested tool:	pwntools, shell + python, gdb.
Hints:	use, again, a `system` function
Source:	0x00sec.org

This time there is no useful function, implemented by the user, to jump to. But there is plenty of functions inside the `libc`.

Try to craft your ROP chain in two ways:

- put the argument of the `system` inside the buffer and use its address to point at what you wrote
- search (e.g. with gdb) for the right argument inside the binary

The binary file is available in the lab03 material file:
`/lab03_material/rop/rop3`
The source code is also available in the material file:
`/lab03_material/rop/rop3.c`

Exercise 4

Purpose:	spawn a shell exploiting ROP.
Suggested tool:	pwntools, shell + python, gdb.
Hints:	exploit a logic flaw to pass the canary check
Author:	AC (pwnthem0le)

This exercise is very similar to the Exercise 5. The difference is in the stack canary, which is now enabled; this makes overwriting the return address almost impossible.
Be aware that this is the most difficult exercise of this section.
The binary file is available in the lab03 material file:
`/lab03_material/rop/rop3`