OT Lab 2 - Software Testing

In this lab, you will learn how to perform a buffer overflow attack. This is a phenomenon that occurs when a computer program writes data outside of a buffer allocated in memory. A buffer overflow can cause a crash or program hang leading to a denial of service (denial of service). Certain types of overflows, such as stack frame overflows, allow an attacker to load and execute arbitrary machine code on behalf of the program and with the rights of the account from which it is executed and thus gain a root shell.

Buffer Overflow (low-level exploitations)

Task 1 - Theory

1. What binary exploitation mitigation techniques do you know?
2. Did NX solve all binary attacks? Why?
3. Why do stack canaris end with 00?
4. What is NOP sled?

Task 2 - Binary attack warming up

We are going to work on a buffer overflow attack as one of the most popular and widely spread binary attacks. You are given a binary **warm_up.** Link

Answer the question and provide explanation details with PoC: "Why in the **warm_up** binary, opposite to the **binary64** from Lab1, the value of **i** doesn't change even if our input was very long?"

Task 3 - Linux local buffer overflow attack x86

You are given a very simple C code:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
char buf[128];
strcpy(buf, argv[1]);
printf("%s\n", buf);
return 0;
}
```

1. Create a new file and put the code above into it:

```
touch source.c
```

2. Compile the file with C code in the binary with the following parameters in the case if you use x64 system:

```
gcc -o binary -fno-stack-protector -m32 -z execstack source.c
```

Questions:

- What does mean **-fno-stack-protector** parameter?
- What does mean **-m32** parameter?
- What does mean **-z execstack** parameter?

If you use x64 system, install the following package before compiling the program:

```
sudo apt install gcc-multilib
```

3. Disable ASLR before to start the buffer overflow attack:

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

4. Anyway, you can just download the pre-compiled binary task 3.

5. Choose any debugger to disassemble the binary. E.g. GNU debugger (gdb).

6. Perform the disassembly of the *required* function of the program.

7. Find the name and address of the *target* function. Copy the address of the function that follows (*is located below)* this function to jump across EIP.

8. Set the breakpoint with the assigned address.

9. Run the program with output that corresponds to the size of the buffer.

10. Examine the stack location and detect the start memory address of the buffer. In other words, this is the point where we break the program and rewrite the EIP.

11. Find the size of the writable memory on the stack. Re-run the same command as in the step #9 but without breakpoints now. Increase the size of output symbols with several bytes that we want to print until we get the overflow. In this way, we will iterate through different addresses in memory, determine the location of the stack and find out where we can "jump" to execute the shell code. Make sure that you get the segmentation fault instead the normal program completion. In simple words, we perform a kinda of *fuzzing*.

12. After detecting the size of the writable memory on the previous step, we should figure out the **NOP sleds** and inject out shell code to fill this memory space. You can find the shell codes examples on the external resources, generate it by yourself (e.g., msfvenom).
You are also given the pre-prepared 46 bytes shell code:

```
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

The shell code address should lie down after the address of the return function.

13. Basically, we don't know the address of the shell code. We can avoid this issue using **NOP** processor instruction: **0x90 .** If the processor encounters a **NOP** command, it simply proceeds to the next command (on next byte). We can add many **NOP** sleds and it helps us to execute the shell code regardless of overwriting the return address.
Define how many **NOP** sleds you can write: *Value of the writable memory - Size of the shell code.*

14. Run the program with our exploit composition:

```
\x90 * the number of NOP sleds + shell code + the memory location
```

that we want to "jump" to execute our code. To do it, we have to overwrite the IP which prescribe which piece of code will be run next.
Remark: \x90 is is a NOP instruction in Assembly.

15. Make sure that you get the root shell on your virtual machine.

Bonus. Answer to the question: "It is possible that sometimes with the above binary shell is launched under gdb, but if you just run the program, it crashes with segfault. Explain why this is happening."