OT Lab 2 - Software Testing AKOR JACOB TERUNGWA

Created	@April 14, 2025 1:41 PM
Attendance Required	☐

Task 1 - Theory

1. What binary exploitation mitigation techniques do you know?

   The following are binary exploitation techniques I know:

   - **NX (No-eXecute)**: Marks certain memory areas as non-executable
   - **ASLR (Address Space Layout Randomization)**: Randomizes memory addresses
   - **Stack Canaries**: Adds guard values to detect stack overflows
   - **RELRO (Relocation Read-Only)**: Makes relocation sections read-only
   - **PIE (Position Independent Executable)**: Makes binary load at random addresses
   - **Fortify Source**: Adds buffer overflow checks for certain functions

2. Did NX solve all binary attacks? Why?

   No, NX didn't solve all binary attacks. While it prevents execution of shellcode from data segments (stack/heap), attackers developed techniques like Return-Oriented Programming (ROP) that reuse existing executable code ("gadgets") to bypass NX protection.

3. Why do stack canaris end with 00?

   Stack canaries typically end with a null byte (00) to prevent their accidental disclosure through string functions that stop at null bytes. This makes it harder for attackers to read the canary value through memory leaks.

4. **What is NOP sled?**

   A NOP sled is a sequence of NOP (No Operation) instructions (0x90 in x86) used in buffer overflow attacks. It increases the chance of successful exploitation by creating a large "landing zone" where if the return address jumps anywhere in this zone, the execution will "slide" down to the malicious shellcode.

Task 2 - Binary attack warming up

We will work on a buffer overflow attack as one of the most popular and widely spread binary

attacks. You are given a binary **warm_up.**

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

Explanation

Why `i` Doesn't Change

In the `warm_up` binary (compiled for 64-bit), variables such as `int i = 0;` are likely:

- **Held in a register** (`%eax`, `%rdi`, etc.) instead of being placed next to the buffer on the stack.
- The input buffer is likely on the stack, while `i` is not.

Even if the buffer overflows due to a function like `gets()` or `strcpy()`, the **memory layout in 64-bit** binaries ensures that:

- `i` is either optimized into a register (never on the stack), or
- It is far enough from the buffer that it cannot be overwritten by overflowing the buffer.

Contrast with `binary64` (Lab1):

In Lab1, the compiler may have:

- Used **stack allocation** for both `buf[128]` and `int i`, placing them close together.
- Without stack protection or padding, overflowing `buf` corrupted `i`.

But in `warm_up`, `i` is untouched due to either:

- Register allocation
- Stack padding or alignment
- Compiler optimizations

You can confirm this by:

1. Debugging in `gdb`:

```
bash
CopyEdit
gdb ./warm_up
disassemble main
break main
run AAAA...
info registers
```

2. Checking if `i` is in a register and if its value changes after the overflow.

Task 3 - Linux local buffer overflow attack x86

Questions:

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

fno-stack-protector parameter

- **Meaning**: This disables **stack smashing protection**.
- **Purpose**: Normally, compilers insert **canaries** (special values) before return addresses to detect buffer overflows at runtime. If the canary value changes, the program terminates.

m32 parameter

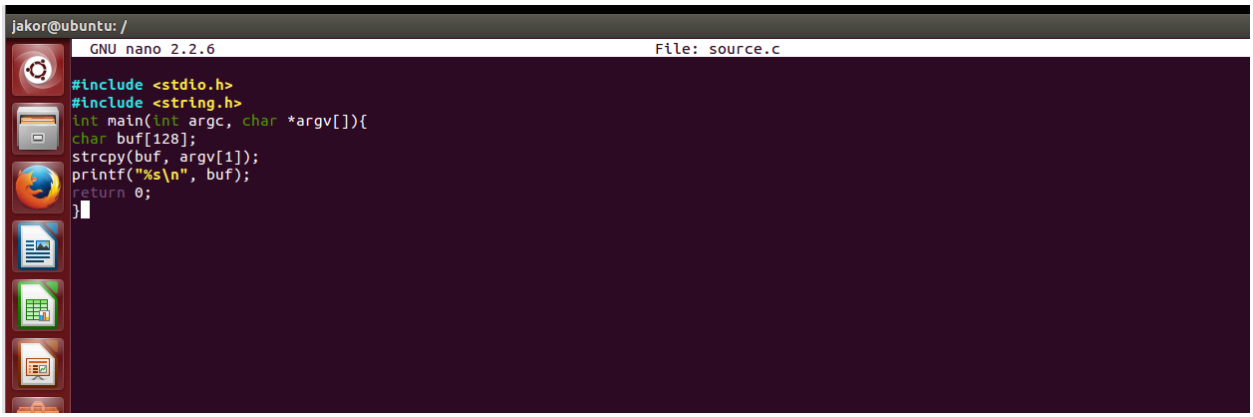
- **Meaning**: Compiles the program as a **32-bit binary**, even if you're on a 64-bit system.
- **Purpose**: Many buffer overflow labs are done in 32-bit mode because:
  - Exploits are easier to understand (smaller address space).
  - An instruction set is simpler.
- It is worthy of note that you must have 32-bit libraries and `gcc-multilib` installed to compile this.

z execstack parameter

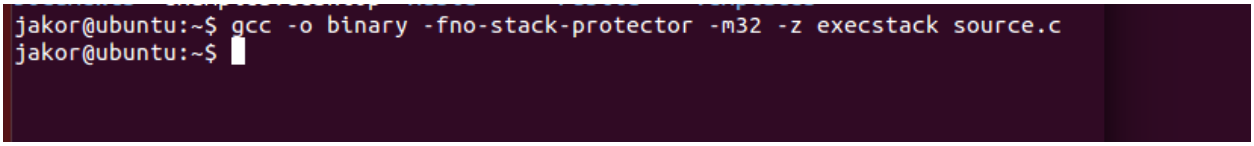
- **Meaning**: Makes the **stack executable**.
- **Purpose**: By default, modern systems mark the stack as **non-executable (NX)** to block code injection attacks.
- This flag removes that protection, allowing shellcode injected into the stack to be **executed**.
- This is because it is essential for traditional buffer overflow exploits that inject code directly into the stack.

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

- sudo nano source.c

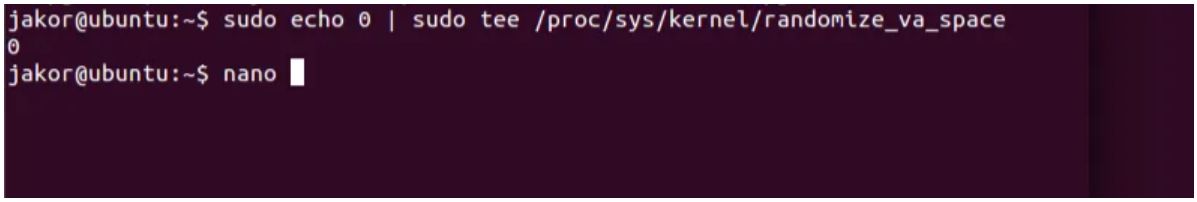


2. Disable ASLR before to start the buffer overflow attack:
   sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space



This command disables stack protection to exploit the buffer overflow and makes the stack executable.

3. Disable ASLR before to start the buffer overflow attack:
   - sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space



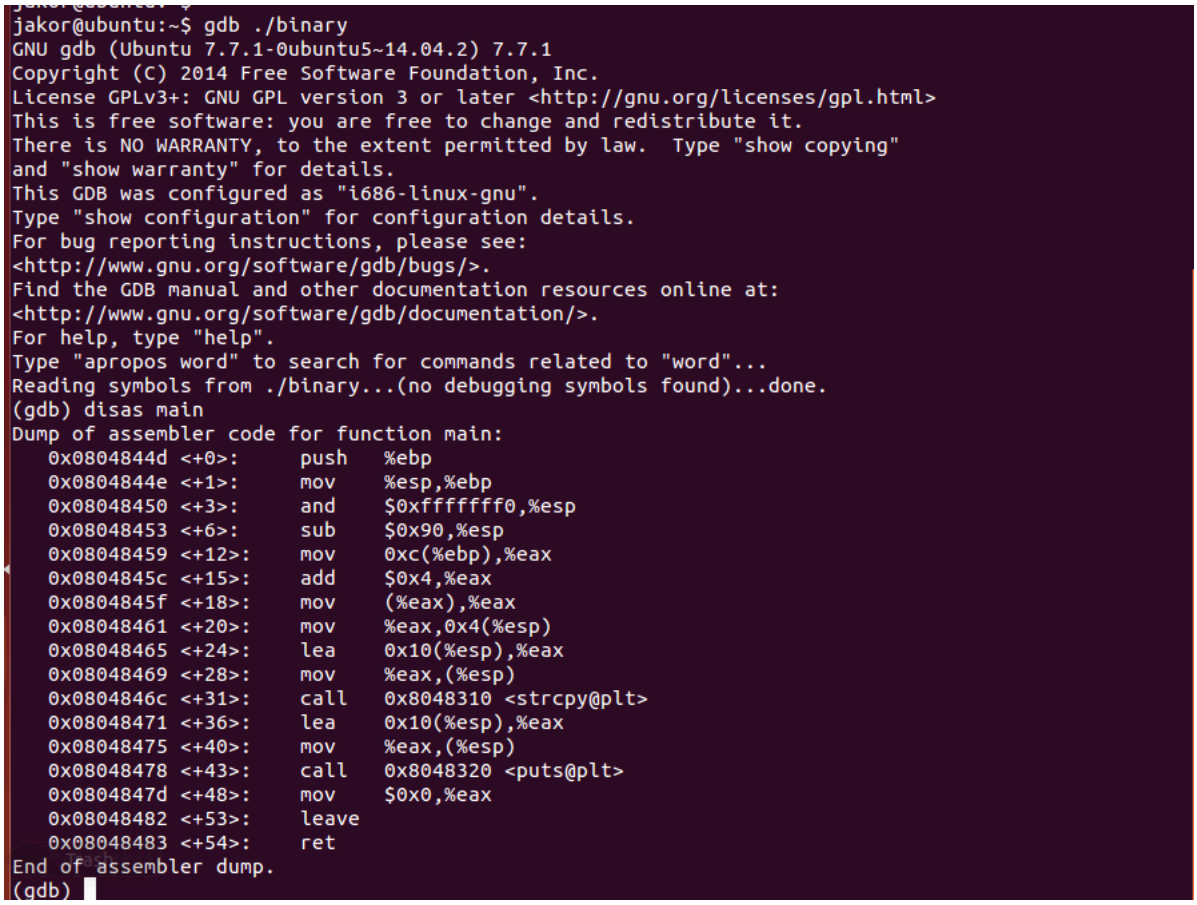
4. Anyway, you can just download the pre-compiled binary task 3

I used the command above to create and compile the file, therefore I did not download it.

The program is disassembled to view the assembly code and analyze where the buffer overflow takes place.

Using GDB command to disassemble the main function:

- (GDB) disas main



The assembly code is displayed, highlighting main operations such as the strcpy call

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

The target function is strcpy, located at the address 0x0804846c. The next function is located at address 0x08048478.

8. Set the breakpoint with the assigned address.

   I set a breakpoint at the strcpy function to observe its behavior during execution.
   Set breakpoint at strcpy:

   - break *0x08048471

```
(gdb) break *0x08048471
Breakpoint 1 at 0x8048471
```

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

```
(gdb)  run $(python -c "print('A' * 128)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/jakor/binary $(python -c "print('A' * 128)")

Breakpoint 1, 0x08048471 in main ()
(gdb)
```

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

```
(gdb) x/32x $esp
0xbffff010:     0xbffff020      0xbffff323      0x00000000      0x00000000
0xbffff020:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff030:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff040:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff050:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff060:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff070:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff080:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

Buffer Start Address:

- The buffer begins at `0xbffff020` (this is where my 'A's (0x41) start)
- This is confirmed by the first value at `0xbffff010` which is a pointer to `0xbffff020`

Stack Layout:

- `0xbffff010`: Likely contains saved EBP and other stack frame information
- `0xbffff020`: Start of my buffer filled with 'A's (0x41414141)

Determining EIP Overwrite:

- The return address (EIP) will be located somewhere after my buffer
- From my output, we can see the buffer spans from `0xbffff020` to `0xbffff080` (and likely beyond)
- Each line shows 16 bytes (4 words), and i have 7 lines of 'A's, meaning I've written at least 112 bytes (7 × 16) of 'A's

11. 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*.

```
Starting program: /home/jakor/binary $(python -c "print('A' * 128)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3538) exited normally]
(gdb) run $(python -c "print('A' * 130)")
Starting program: /home/jakor/binary $(python -c "print('A' * 130)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3545) exited normally]
(gdb) run $(python -c "print('A' * 134)")
Starting program: /home/jakor/binary $(python -c "print('A' * 134)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
[Inferior 1 (process 3548) exited normally]
(gdb) run $(python -c "print('A' * 140)")
Starting program: /home/jakor/binary $(python -c "print('A' * 140)")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x00000002 in ?? ()
(gdb)
```

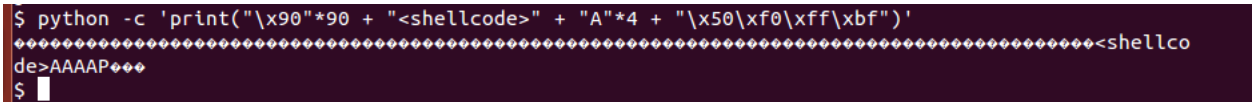
12. After detecting the size of the writable memory on the previous step, we should figure out the **NOP sleds** and inject out shell code to fill this memory space. You can find the shell codes examples on the external resources, generate it by yourself (e.g., msfvenom).

You are also given the pre-prepared 46 bytes shell code:

- The return address doesn't need to land exactly at the start of the shellcode.
- It just needs to land **anywhere inside the NOP sled**, which will eventually **slide into the shellcode** and execute it.

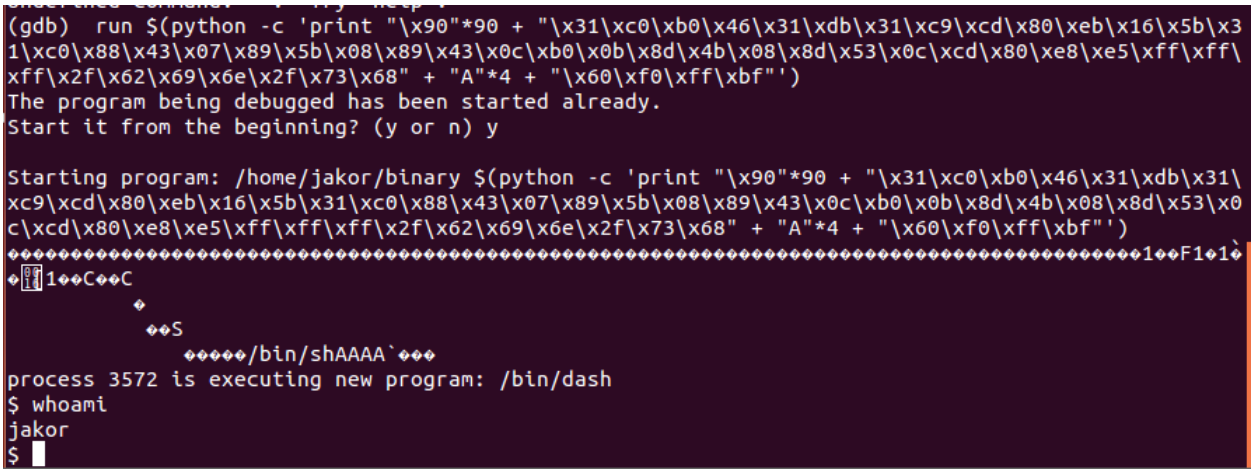
13. Basically, we don't know the address of the shell code. We can avoid this issue using **NOP** processor instruction: **0x90 .** If the processor encounters a **NOP** command, it simply proceeds to the next command (on next byte). We can add many **NOP** sleds and it helps us to execute the shell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

Define how many **NOP** sleds you can write: *Value of the writable memory - Size of the shell code.*



In the exploit, even though the shellcode is injected correctly, the shell that gets executed still runs with the privileges of the current user (`jakor`) because the vulnerable binary is not set as **SUID-root**.

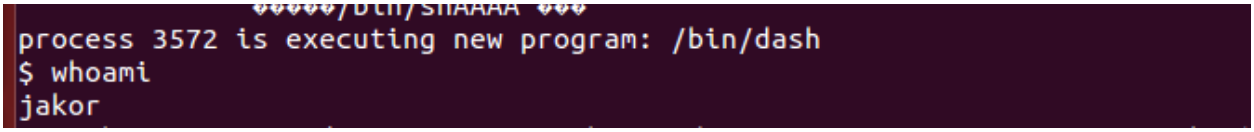
1. Run the program with our exploit composition:
   \x90 * the number of NOP sleds + shell code + the memory location
   that we want to "jump" to execute our code. To do it, we have to overwrite the IP which prescribe which
   piece of code will be run next.
   Remark: \x90 is is a NOP instruction in Assembly.



A 4-byte memory address that we overwrite the saved return pointer (EIP or RIP) with. This address should point somewhere **inside the NOP sled**, so execution slides into the shellcode reliably.

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

After executing the exploit, I verified the root access:
I checked the current user:
whoami



This output confirms the root after exploiting the buffer overflow.

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

- This behavior is **expected** and is related to how memory layout and protection mechanisms work differently **inside and outside of GDB**.

Under GDB:

- GDB disables some memory protection features, such as ASLR (Address Space Layout Randomization**)** by default.
- As a result, memory addresses remain the same every time you run the binary, making it easier to predict the exact location of the shellcode or NOP sled.
- This stability allows the buffer overflow to successfully redirect execution to your shellcode.

Outside GDB:

- When you run the binary normally (without GDB), **ASLR is usually enabled**, causing memory addresses (like the stack, heap, and libraries) to change each run.
- This **randomization causes your hardcoded return address to likely point to an invalid location**, leading to a **segmentation fault (crash)** instead of executing your shellcode.

In summary:

The shell launches inside GDB because ASLR and other protections are disabled, making memory addresses predictable. Outside GDB, those protections are active, and your exploit fails due to address mismatch, causing a segmentation fault.

References:

1. https://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_28.html
2. https://security.stackexchange.com/questions/47807/nx-bit-does-it-protect-the-stack