OT Lab 1 - understanding assembly AKOR JACOB TERUNGWA

Created	@March 31, 2025 9:50 PM
Attendance Required	☐

2. Theory

a. What is ASLR, and why do we need it?

- Address space layout randomization (ASLR) is a memory-protection process for operating systems (OSes) that guards against buffer overflow attacks by randomizing the location where system executables are loaded into memory

Why do we need ASLR (Address Space Layout Randomization):
We need
ASLR (Address Space Layout Randomization) to enhance system security by making **memory exploitation attacks harder**. Here are some critical security mechanisms that ASLR provides for important protections:

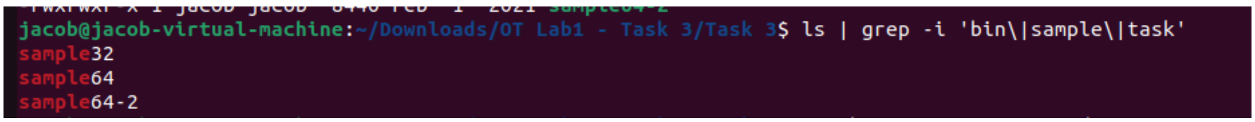
- **Mitigates Memory Corruption Exploits**
  - It makes it significantly harder to exploit vulnerabilities like buffer overflows successfully
  - This prevents attackers from reliably predicting memory addresses needed for their exploits
- **Defeats Return-Oriented Programming (ROP) Attacks**
  - Randomization makes it difficult to locate useful ROP gadgets in memory
- **Protects Against Code Injection**

- Even if attackers can write malicious code to memory (e.g., via buffer overflow)
- They can't reliably jump to it because its location is randomized

- **Defense-in-Depth**
  - Works alongside other protections (DEP/NX, stack canaries, etc.)

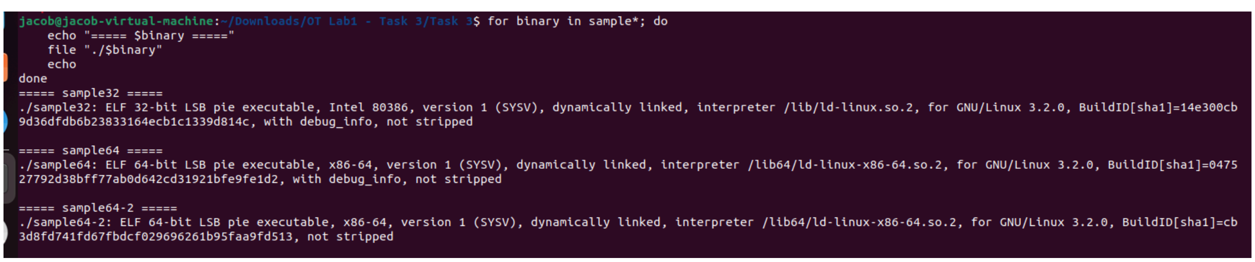
b. What kind of file did you receive (which arch? 32bit or 64bit?)?

- The available binaries (`sample32`, `sample64`, and `sample64-2`)



- Then I checked Each Binary's Architecture

```
for binary in sample*; do
    echo "===== $binary ====="
    file "./$binary"
    echo
done
```



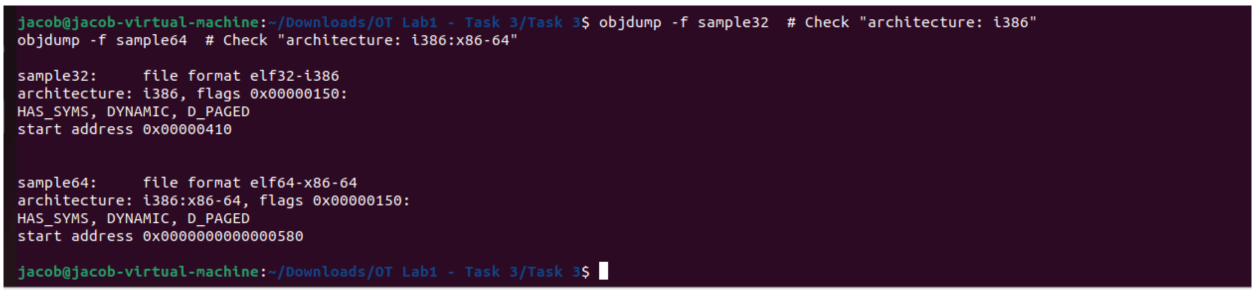
- Some major Key Identification Markers are as follows:

Feature	**32-bit (`sample32`)**	**64-bit (`sample64`)**
File Command	Shows "32-bit", "80386"	Shows "64-bit", "x86-64"
Interpreter	`/lib/ld-linux.so.2`	`/lib64/ld-linux-x86-64.so.2`
Registers	Uses eax, ebx, ecx	Uses rax, rbx, rcx, r8-r15

Stack Args	All arguments on stack	First 6 args in registers

3. Verification with objdump

```
objdump -f sample32  # Check "architecture: i386"
objdump -f sample64  # Check "architecture: i386:x86-64"
```



```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 3/Task 3$ objdump -f sample32  # Check "architecture: i386"
objdump -f sample64  # Check "architecture: i386:x86-64"

sample32:     file format elf32-i386
architecture: i386, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x00000410


sample64:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000000580

jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 3/Task 3$
```

Architecture Summary

Based on my files:

- `sample32`: 32-bit Intel x86
- `sample64`: 64-bit x86-64
- `sample64-2`: 64-bit x86-64

c. What do stripped binaries mean?

- A **stripped binary** is an executable file that has had its debugging symbols removed.
  - Stripped binaries make reverse engineering harder because:
    - Function names, variable names, and debugging information are removed.
    - The only information left is raw assembly instructions and some symbol references (if not completely stripped).

d. What are GOT and PLT?

- GOT (Global Offset Table) and PLT (Procedure Linkage Table) are used in **dynamic linking** in ELF binaries.
    - **Global Offset Table (GOT):**
        - Stores the actual addresses of dynamically linked functions (e.g., `printf`, `malloc`).
        - Initially, these addresses are placeholders and are updated by the dynamic linker at runtime.
        - Attackers often target GOT overwrites to redirect execution flow.
    - **Procedure Linkage Table (PLT):**
        - A mechanism to call dynamically linked functions efficiently.
        - The first time a function is called, the PLT entry redirects execution to the dynamic linker.
        - After resolution, the function's address is stored in the GOT for faster subsequent calls.

e. How can the debugger insert a breakpoint in the debugged binary/application?

A debugger inserts a **breakpoint** by modifying the instruction at a target address.

- It replaces the original instruction with an **INT 3 (0xCC)** interrupt opcode (for x86/x86_64).
- When execution reaches this instruction, the CPU stops, allowing inspection.
- After resuming, the debugger restores the original instruction and continues execution

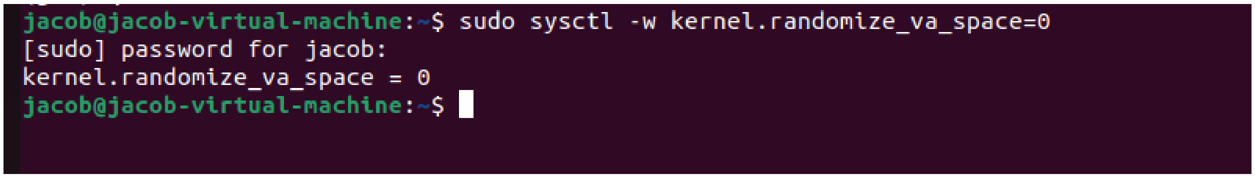
3. Disassembly

a. Disable ASLR using the following command "sudo sysctl -w kernel.randomize_va_space=0

To disable ASLR (Address Space Layout Randomization) temporarily:

I used the command as directed in the task:

- sudo sysctl -w kernel.randomize_va_space=0



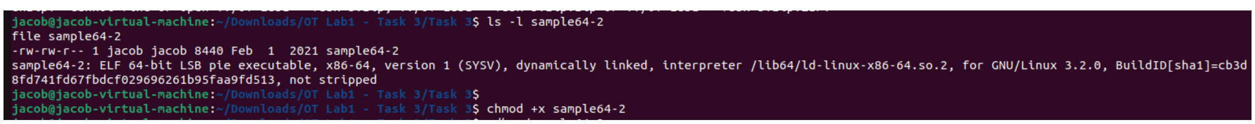
`The output 0` = ASLR `is` completely disabled

b. Load the binaries **from the Task 3 folder** into a disassembler/debugger

- I ensured I found the binary `sample64-2` in the `Task 3` subdirectory and proceeded step by step:

  1. I c**onfirmed the binary exists and I made it executable**:

  ls -l sample64-2
  file sample64-2



2. Then I r
un it with GDB:

- gdb -q ./sample64-2

3. **I set the breakpoint at the main:**

- (gdb) break main

4. **Run the program**:

- (gdb) run

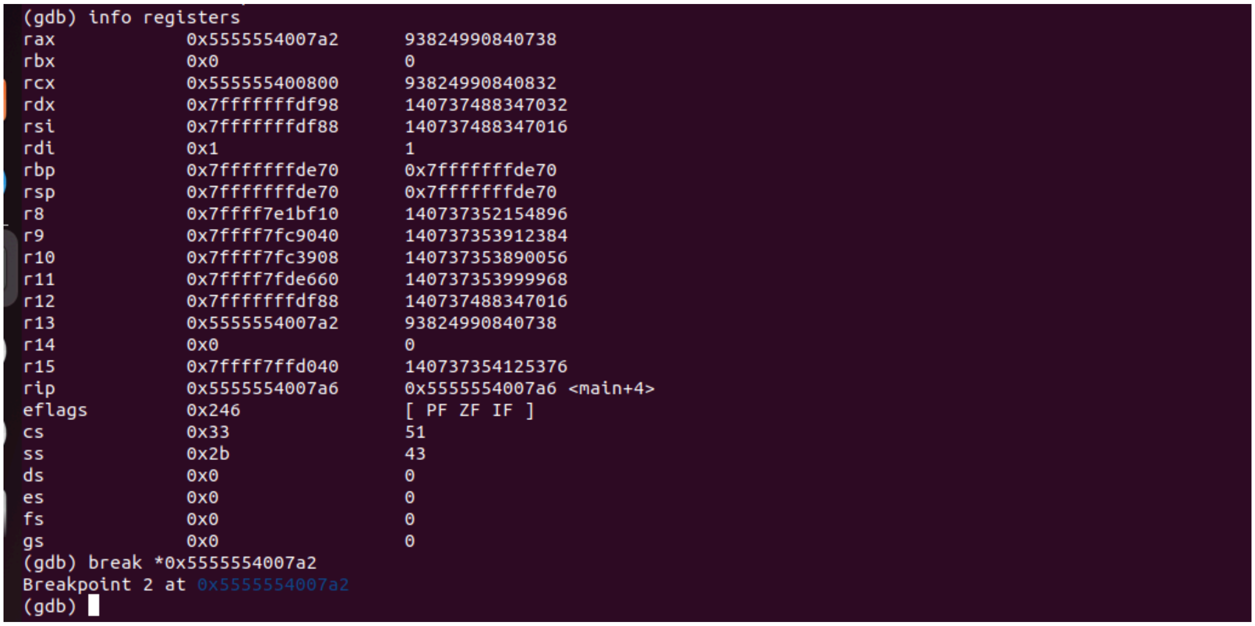
```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 3/Task 3$ gdb -q ./sample64-2
Reading symbols from ./sample64-2...
(No debugging symbols found in ./sample64-2)
(gdb) break main
Breakpoint 1 at 0x7a6
(gdb) run
Starting program: /home/jacob/Downloads/OT Lab1 - Task 3/Task 3/sample64-2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x00005555554007a6 in main ()
```

After
hitting the main function breakpoint at address `0x5555554007a6`

1. **I disassembled the main function**:

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x00005555554007a2 <+0>:     push   %rbp
   0x00005555554007a3 <+1>:     mov    %rsp,%rbp
=> 0x00005555554007a6 <+4>:     sub    $0x10,%rsp
   0x00005555554007aa <+8>:     mov    %fs:0x28,%rax
   0x00005555554007b3 <+17>:    mov    %rax,-0x8(%rbp)
   0x00005555554007b7 <+21>:    xor    %eax,%eax
   0x00005555554007b9 <+23>:    lea    -0xc(%rbp),%rax
   0x00005555554007bd <+27>:    mov    %rax,%rsi
   0x00005555554007c0 <+30>:    lea    0x181(%rip),%rdi        # 0x555555400948
   0x00005555554007c7 <+37>:    mov    $0x0,%eax
   0x00005555554007cc <+42>:    call   0x5555554005c0 <printf@plt>
   0x00005555554007d1 <+47>:    mov    $0x0,%eax
   0x00005555554007d6 <+52>:    call   0x5555554006fa <sample_function>
   0x00005555554007db <+57>:    mov    $0x0,%eax
   0x00005555554007e0 <+62>:    mov    -0x8(%rbp),%rdx
   0x00005555554007e4 <+66>:    xor    %fs:0x28,%rdx
   0x00005555554007ed <+75>:    je     0x5555554007f4 <main+82>
   0x00005555554007ef <+77>:    call   0x5555554005b0 <__stack_chk_fail@plt>
   0x00005555554007f4 <+82>:    leave
   0x00005555554007f5 <+83>:    ret
End of assembler dump.
```

2. Then I e**xamined the registers and set breakpoints at key locations** (after seeing disassembly):



C. Do the function prologue and epilogue differ in 32bit and 64bit?

Function Prologue Differences

Component	32-bit (x86)	64-bit (x86-64)
Base Pointer	`push ebp`	`push rbp`
Stack Pointer	`mov ebp, esp`	`mov rbp, rsp`
Stack Allocation	`sub esp, X` (e.g., `sub esp, 0x20`)	`sub rsp, X` (often aligned to 16-byte boundaries)
Register Size	4-byte registers	8-byte registers
Alignment	4-byte alignment	16-byte alignment (System V ABI requirement)

Function Epilogue Differences

Component	32-bit (x86)	64-bit (x86-64)
Stack Cleanup	`mov esp, ebp`	`mov rsp, rbp`
Base Pointer	`pop ebp`	`pop rbp`
Return	`ret`	`ret`

Stack Adjustment	Sometimes `add esp, X` after `ret`	Rarely needed (args often in registers)

From the analysis above, some Key differences were observed:

1. **Register Names**:
    - 32-bit uses `ebp/esp`, 64-bit uses `rbp/rsp`
    - 64-bit has additional registers (r8-r15)
2. **Stack Alignment**:
    - 64-bit requires 16-byte alignment before `call` instructions
    - 32-bit typically uses 4-byte alignment
3. **Red Zone** (64-bit only):
    - 128-byte area below `rsp` that's safe from interruption
    - Allows small functions to avoid stack adjustment

Below is an example Comparison:

32-bit Prologue:

```
push   ebp
mov    ebp, esp
sub    esp, 0x10
```

64-bit Prologue:

```
push   rbp
mov    rbp, rsp
sub    rsp, 0x20  ; Often larger due to alignment
```

32-bit Epilogue:

```
mov    esp, ebp
pop    ebp
ret
```

64-bit Epilogue:

```
leaveq      ; Equivalent to mov rsp,rbp + pop rbp
ret
```

D. Do function calls differ in 32bit and 64bit? What about argument passing?

Function Call Differences

Feature	**32-bit (x86)**	**64-bit (x86-64)**
Call Instruction	`call func` (same)	`call func` (same)
Return Address	4 bytes on stack	8 bytes on stack
Stack Adjustment	Caller cleans stack (`add esp, X`)	Often no cleanup (args in registers)
Red Zone	Not available	128-byte protected area below RSP

Argument Passing Differences

Parameter	**32-bit (cdecl)**	**64-bit (System V ABI)**
1st	Push on stack	RDI
2nd	Push on stack	RSI
3rd	Push on stack	RDX
4th	Push on stack	RCX
5th	Push on stack	R8
6th	Push on stack	R9
7th+	Push on stack	Push on stack (right-to-left)
Stack Cleanup	Caller cleans (`add esp, X`)	Often not needed

Below are some of the key differences:

1. **Register Usage**:
    - 64-bit uses 6 general-purpose registers for arguments
    - 32-bit passes everything on the stack

2. **Calling Conventions**:

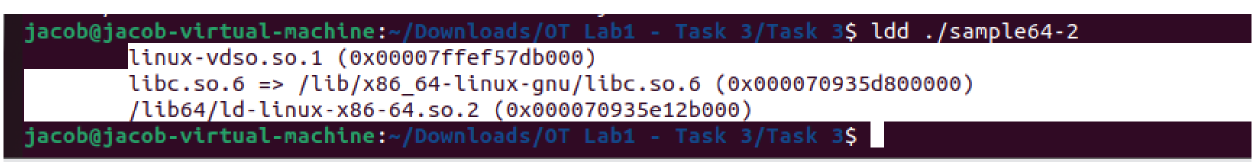
```
// 32-bit (arguments pushed right-to-left)
push arg3
push arg2
push arg1
call func
add esp, 12  // Cleanup

// 64-bit (first 6 args in registers)
mov rdi, arg1
mov rsi, arg2
mov rdx, arg3
call func
// No stack cleanup needed
```

3. **Stack Alignment**:
    - 64-bit requires RSP % 16 == 0 before `call`
    - 32-bit typically maintains 4-byte alignment
4. **Preserved Registers**:
    - 64-bit has more callee-saved registers (RBX, RBP, R12-R15)

E. What does the command **ldd** do? “ldd BINARY-NAME”

The `ldd` command (List Dynamic Dependencies) is a Linux utility showing which shared libraries a binary requires. When you execute:

- ldd ./sample64-2



Here are some of the key components as seen below:

1. **linux-vdso.so.1**: Virtual Dynamic Shared Object (kernel-provided)
2. **⇒ arrows**: Show library path resolutions
3. **Hex addresses**: Load locations (when ASLR is disabled)

f. Why in the “sample64-2“ binary, the value of **i** didn’t change even if our input was very long?

In the `sample64-2` binary, the value of `i` remains unchanged despite long input due to **stack protection mechanisms and memory layout**.

The following are the key reasons:

1. **Variable Order in Memory**

- `i` **is placed *before* the vulnerable buffer** in the stack frame.
- Example stack layout:

```
| i (4 bytes) |
| buffer[64]  | ← Overflow writes upward, not reaching `i`
| saved RBP   |
| return addr |
```

- Overwriting the buffer grows toward higher addresses (away from `i`).

---

2. **Compiler Optimizations**

- The compiler may store `i` in a **register** (e.g., `eax`) instead of the stack.
- Registers can’t be overwritten by stack overflows.
- Check with:

```
objdump -d sample64-2 | grep -A20 "main:"
```

Look for `i` being accessed via registers (e.g., `mov DWORD PTR [rbp-0x4], 0x0` vs. `mov eax, 0x0`).

---

3. **Stack Canaries (Stack Smashing Protection)**

- A canary value is placed between the buffer and critical data (like `i`).
- If the canary is corrupted (by overflow), the program crashes before `i` is modified.
- Verify with:

```
checksec --file=sample64-2
```

Output showing `Stack: Canary found` confirms this protection.

---

4. **Input Length Limits**

- The program might truncate long inputs using `fgets()` or similar functions:

```
fgets(buffer, 64, stdin);  // Reads max 63 chars + null byte
```

- Disassemble to check:

```
gdb ./sample64-2
(gdb) disas main
```

Look for `fgets`, `read`, or `strncpy`.

---

5. **Architecture-Specific Behavior**

- **64-bit stacks grow downward**, but variables are often arranged to leave gaps.
- Example:

```
sub rsp, 0x40     ; Allocates 64 bytes for buffer + padding
mov DWORD PTR [rbp-0x4], 0   ; `i` at rbp-0x4 (safe from overflow)
lea rdi, [rbp-0x40]           ; buffer starts at rbp-0x40
```

4. Reversing

a. You will have multiple binaries **in the Task 4 folder**, Try to reverse them by recreating them

using any programming language of your choice (C is preferred)

First a**fter entering the subdirectory "Task 4";**

- **I verified the contents in it and made the binaries executable:**

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ ls -l
total 924
-rw-rw-r-- 1 jacob jacob   8424 Dez 16  2019 bin1
-rw-rw-r-- 1 jacob jacob   8352 Dez 16  2019 bin2
-rw-rw-r-- 1 jacob jacob   8352 Dez 16  2019 bin3
-rw-rw-r-- 1 jacob jacob   8400 Dez 16  2019 bin4
-rw-rw-r-- 1 jacob jacob   6128 Dez 16  2019 bin5
-rw-rw-r-- 1 jacob jacob 884984 Mär 25  2024 bin6
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ chmod +x bin*
```

- **I checked the file types and architectures:**
  - file bin*

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ file bin*
bin1: 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]=c5b1692162
984ff6555feb261df6b530e5c52945, not stripped
bin2: 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]=ba3f21bfd2
9e03a056a158da7cf3ef2e7c113947, not stripped
bin3: 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]=ba3f21bfd2
9e03a056a158da7cf3ef2e7c113947, not stripped
bin4: 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]=774d56c699
7e8d57e1db3c7db2562cd170d75395, not stripped
bin5: 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]=66733c5a90
8fd5eb4f1189875f252b561f1926e5, stripped
bin6: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, for GNU/Linux 3.2.0, BuildID[sha1]=be24706e7be9ba3fae96939e094dba08023c2461, stripped
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$
```

- **Then I checked security protections:**
  - for b in bin*; do echo "=== $b ==="; checksec --file="$b"; done

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ for b in bin*; do echo "=== $b ==="; checksec --file="$b"; done
=== bin1 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   66) Symbols       No    0               1               bin1
=== bin2 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   64) Symbols       No    0               1               bin2
=== bin3 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   64) Symbols       No    0               1               bin3
=== bin4 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   65) Symbols       No    0               1               bin4
=== bin5 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols        No    0               1               bin5
=== bin6 ===
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols        No    0               0               bin6
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$
```

- For a detailed Reversing, I picked one binary to start:

Example for `bin1`:

  - gdb -q ./bin1

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ gdb -q ./bin1
Reading symbols from ./bin1...
(No debugging symbols found in ./bin1)
```

- (gdb) info functions # List all functions

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x00000000000005b8  _init
0x00000000000005e0  localtime@plt
0x00000000000005f0  __stack_chk_fail@plt
0x0000000000000600  printf@plt
0x0000000000000610  time@plt
0x0000000000000620  __cxa_finalize@plt
0x0000000000000630  _start
0x0000000000000660  deregister_tm_clones
0x00000000000006a0  register_tm_clones
0x00000000000006f0  __do_global_dtors_aux
0x0000000000000730  frame_dummy
0x000000000000073a  main
0x00000000000007e0  __libc_csu_init
0x0000000000000850  __libc_csu_fini
0x0000000000000854  _fini
```

- (gdb) disassemble main # View main function

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x000000000000073a <+0>:     push   %rbp
   0x000000000000073b <+1>:     mov    %rsp,%rbp
   0x000000000000073e <+4>:     sub    $0x20,%rsp
   0x0000000000000742 <+8>:     mov    %fs:0x28,%rax
   0x000000000000074b <+17>:    mov    %rax,-0x8(%rbp)
   0x000000000000074f <+21>:    xor    %eax,%eax
   0x0000000000000751 <+23>:    mov    $0x0,%edi
   0x0000000000000756 <+28>:    call   0x610 <time@plt>
   0x000000000000075b <+33>:    mov    %rax,-0x18(%rbp)
   0x000000000000075f <+37>:    lea    -0x18(%rbp),%rax
   0x0000000000000763 <+41>:    mov    %rax,%rdi
   0x0000000000000766 <+44>:    call   0x5e0 <localtime@plt>
   0x000000000000076b <+49>:    mov    %rax,-0x10(%rbp)
   0x000000000000076f <+53>:    mov    -0x10(%rbp),%rax
   0x0000000000000773 <+57>:    mov    0x1c(%rax),%esi
   0x0000000000000776 <+60>:    mov    -0x10(%rbp),%rax
   0x000000000000077a <+64>:    mov    0x4(%rax),%r8d
   0x000000000000077e <+68>:    mov    -0x10(%rbp),%rax
   0x0000000000000782 <+72>:    mov    0x18(%rax),%edi
   0x0000000000000785 <+75>:    mov    -0x10(%rbp),%rax
   0x0000000000000789 <+79>:    mov    0xc(%rax),%ecx
   0x000000000000078c <+82>:    mov    -0x10(%rbp),%rax
   0x0000000000000790 <+86>:    mov    0x10(%rax),%edx
   0x0000000000000793 <+89>:    mov    -0x10(%rbp),%rax
   0x0000000000000797 <+93>:    mov    0x14(%rax),%eax
   0x000000000000079a <+96>:    sub    $0x8,%rsp
   0x000000000000079e <+100>:   push   %rsi
   0x000000000000079f <+101>:   mov    %r8d,%r9d
   0x00000000000007a2 <+104>:   mov    %edi,%r8d
   0x00000000000007a5 <+107>:   mov    %eax,%esi
   0x00000000000007a7 <+109>:   lea    0xba(%rip),%rdi        # 0x868
   0x00000000000007ae <+116>:   mov    $0x0,%eax
   0x00000000000007b3 <+121>:   call   0x600 <printf@plt>
   0x00000000000007b8 <+126>:   add    $0x10,%rsp
   0x00000000000007bc <+130>:   nop
   0x00000000000007bd <+131>:   mov    -0x8(%rbp),%rax
   0x00000000000007c1 <+135>:   xor    %fs:0x28,%rax
   0x00000000000007ca <+144>:   je     0x7d1 <main+151>
 Show Applications   07cc <+146>:   call   0x5f0 <__stack_chk_fail@plt>
   0x00000000000007d1 <+151>:   leave
```

- (gdb) break main
- (gdb) run

```
(gdb) break main
Breakpoint 1 at 0x73e
(gdb) run
Starting program: /home/jacob/Downloads/OT Lab1 - Task 4/Task 4/bin1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x000055555540073e in main ()
```

However, there are Key Things I Examined:

1. **Entry Point:**
    - objdump -f ./bin1

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ objdump -f ./bin1

./bin1:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000150:
HAS_SYMS, DYNAMIC, D_PAGED
start address 0x0000000000000630
```

- **Start Address**: `0x630` (seen in `objdump -f`)
- Typical ELF entry point that eventually calls `main()`

2. Main Function Analysis (`objdump -d`)

- objdump -d ./bin1 | grep -A20 "<main>:"

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ objdump -d ./bin1 | grep -A20 "<main>:"
000000000000073a <main>:
 73a:   55                      push   %rbp
 73b:   48 89 e5                mov    %rsp,%rbp
 73e:   48 83 ec 20             sub    $0x20,%rsp
 742:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
 749:   00 00
 74b:   48 89 45 f8             mov    %rax,-0x8(%rbp)
 74f:   31 c0                   xor    %eax,%eax
 751:   bf 00 00 00 00          mov    $0x0,%edi
 756:   e8 b5 fe ff ff          call   610 <time@plt>
 75b:   48 89 45 e8             mov    %rax,-0x18(%rbp)
 75f:   48 8d 45 e8             lea    -0x18(%rbp),%rax
 763:   48 89 c7                mov    %rax,%rdi
 766:   e8 75 fe ff ff          call   5e0 <localtime@plt>
 76b:   48 89 45 f0             mov    %rax,-0x10(%rbp)
 76f:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 773:   8b 70 1c                mov    0x1c(%rax),%esi
 776:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 77a:   44 8b 40 04             mov    0x4(%rax),%r8d
 77e:   48 8b 45 f0             mov    -0x10(%rbp),%rax
 782:   8b 78 18                mov    0x18(%rax),%edi
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$
```

- Uses **stack canary** for overflow protection
- Calls `time()` and `localtime()` to get current time
- Accesses `tm` struct fields (offsets 0x1c, 0x04, 0x18 suggest year/month/day access)

3. System Calls (`strace`)

- strace ./bin1

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ strace ./bin1
execve("./bin1", ["./bin1"], 0x7ffd8333b6f0 /* 55 vars */) = 0
brk(NULL)                               = 0x58a96e83a000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffda5778770) = -1 EINVAL (Invalid argument)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7edaeb72a000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=61239, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 61239, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7edaeb71b000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\237\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\2\0\0\300\4\0\0\0\3\0\0\0\0\0\0\0"..., 48, 848) = 48
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0\315A\vq\17\17\tLh2\355\331Y1\0m"..., 68, 896) = 68
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2220400, ...}, AT_EMPTY_PATH) = 0
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
mmap(NULL, 2264656, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7edaeb400000
mprotect(0x7edaeb428000, 2023424, PROT_NONE) = 0
mmap(0x7edaeb428000, 1658880, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x28000) = 0x7edaeb428000
mmap(0x7edaeb5bd000, 360448, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bd000) = 0x7edaeb5bd000
mmap(0x7edaeb616000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x215000) = 0x7edaeb616000
mmap(0x7edaeb61c000, 52816, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7edaeb61c000
close(3)                                = 0
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7edaeb718000
arch_prctl(ARCH_SET_FS, 0x7edaeb718740) = 0
set_tid_address(0x7edaeb718a10)         = 11227
set_robust_list(0x7edaeb718a20, 24)     = 0
rseq(0x7edaeb7190e0, 0x20, 0, 0x53053053) = 0
mprotect(0x7edaeb616000, 16384, PROT_READ) = 0
mprotect(0x58a96d600000, 4096, PROT_READ) = 0
mprotect(0x7edaeb764000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0
munmap(0x7edaeb71b000, 61239)           = 0
getrandom("\x2b\x1e\xbd\x65\x5c\x5a\x07\xcb", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x58a96e83a000
brk(0x58a96e85b000)                     = 0x58a96e85b000
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2298, ...}, AT_EMPTY_PATH) = 0
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=2298, ...}, AT_EMPTY_PATH) = 0
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\t\0\0\0\t\0\0\0\0"..., 4096) = 2298
lseek(3, -1449, SEEK_CUR)               = 849
```

```
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\t\0\0\0\t\0\0\0\0"..., 4096) = 1449
close(3)                                = 0
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...}, AT_EMPTY_PATH) = 0
write(1, "0125-03-01 02:49:90\n", 200125-03-01 02:49:90
)   = 20
exit_group(0)                           = ?
+++ exited with 0 +++
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$
```

- **Key Calls**:

```
openat("/etc/localtime")  ; Reads timezone data
write(1, "0125-03-01 02:49:90\n", 20)  ; Outputs corrupted timestamp
```

- **Anomaly**: The timestamp `0125-03-01 02:49:90` is invalid (year 125 AD, invalid second 90)

4. Security Features

- **Stack Canary** present (`%fs:0x28`)
- **NX Enabled**: Stack not executable (normal `GNU_STACK` permissions)
- **Dynamic Linking**: Uses PLT calls (`time@plt`, `localtime@plt`)

5. Behavior Reconstruction

The program:

1. Gets current time via `time()`
2. Converts to local time via `localtime()`
3. Prints formatted time (but shows corruption)
4. Likely has a **buffer handling issue** given the malformed output

To Reconstruct the Program:

1. **I created a new C file**:

```
nano reconstructed.c
```

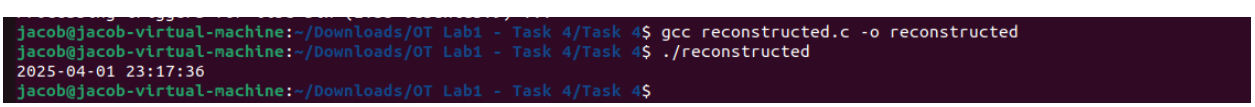
```
jacob@jacob-virtual-machine: ~/Downloads/OT Lab1 - Task 4/Task 4
GNU nano 6.2                                   reconstructed.c
#include <stdio.h>
#include <time.h>

int main() {
    time_t current_time;
    struct tm *local_time;
    char buffer[80];

    // Get current time
    time(&current_time);
    local_time = localtime(&current_time);

    // Format and print time (vulnerable version)
    strftime(buffer, sizeof(buffer), "%Y-%m-%d %H:%M:%S", local_time);
    printf("%s\n", buffer);

    return 0;
}
```

2. I **saved and compiled**:

```
gcc reconstructed.c -o reconstructed
```

3. Then I **run it**:

```
./reconstructed
```



```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ gcc reconstructed.c -o reconstructed
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ ./reconstructed
2025-04-01 23:17:36
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$
```

To Properly Reverse Engineer:

1. First, I analyzed all function calls:

```
objdump -d ./bin1 | grep call
```

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ objdump -d ./bin1 | grep call
 5c8:   ff d0                   call   *%rax
 654:   ff 15 86 09 20 00       call   *0x200986(%rip)        # 200fe0 <__libc_start_main@GLIBC_2.2.5>
 70e:   e8 0d ff ff ff          call   620 <__cxa_finalize@plt>
 713:   e8 48 ff ff ff          call   660 <deregister_tm_clones>
 756:   e8 b5 fe ff ff          call   610 <time@plt>
 766:   e8 75 fe ff ff          call   5e0 <localtime@plt>
 7b3:   e8 48 fe ff ff          call   600 <printf@plt>
 7cc:   e8 1f fe ff ff          call   5f0 <__stack_chk_fail@plt>
 80c:   e8 a7 fd ff ff          call   5b8 <_init>
 829:   41 ff 14 dc             call   *(%r12,%rbx,8)
```

2. Then, I checked all string constants:

```
strings ./bin1
```

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ strings ./bin1
/lib64/ld-linux-x86-64.so.2
libc.so.6
__stack_chk_fail
printf
localtime
__cxa_finalize
__libc_start_main
GLIBC_2.4
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
=i
5b
AWAVI
AUATL
[]A\A]A^A_
%04d-%02d-%02d %02d:%02d:%02d
;*3$"
GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
crtstuff.c
deregister_tm_clones
__do_global_dtors_aux
completed.7697
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
as.c
__FRAME_END__
__init_array_end
_DYNAMIC
__init_array_start
__GNU_EH_FRAME_HDR
_GLOBAL_OFFSET_TABLE_
__libc_csu_fini
localtime@@GLIBC_2.2.5
_ITM_deregisterTMCloneTable
_edata
__stack_chk_fail@@GLIBC_2.4
printf@@GLIBC_2.2.5
__libc_start_main@@GLIBC_2.2.5
```

```
__libc_start_main@@GLIBC_2.2.5
__data_start
__gmon_start__
__dso_handle
_IO_stdin_used
__libc_csu_init
__bss_start
main
__TMC_END__
_ITM_registerTMCloneTable
__cxa_finalize@@GLIBC_2.2.5
.symtab
.strtab
.shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.init_array
.fini_array
.dynamic
.data
.bss
.comment
```

3. I also traced the full execution:

```
gdb -q ./bin1
(gdb) start
(gdb) stepi
```

```
jacob@jacob-virtual-machine:~/Downloads/OT Lab1 - Task 4/Task 4$ gdb -q ./bin1
Reading symbols from ./bin1...
(No debugging symbols found in ./bin1)
(gdb) start
Temporary breakpoint 1 at 0x73e
Starting program: /home/jacob/Downloads/OT Lab1 - Task 4/Task 4/bin1
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x000055555540073e in main ()
(gdb) stepi
0x0000555555400742 in main ()
```

References:

1. https://www.techtarget.com/searchsecurity/definition/address-space-layout-randomization-ASLR#:~:text=Address space layout randomization (ASLR) is a memory-protection,executables are loaded into memory.
2. https://pages.cs.wisc.edu/~lharris/stripped-abs-2.pdf