Laborator 1 – Rezolvare

**Ex 1**. Let's see how a program is loaded into memory from disk. On Linux, executables are usually in the Executable and Linkable Format (ELF). On Windows, the usual format is called Portable Executable (PE). Both have metadata that instructs the operating system how to read and find each part of a program, like data or executable instructions. The data and metadata are kept in \*sections\*. Some of the most common and interesting sections of an ELF file are:

\* .bss -- containing global variables that are uninitialized, or are initialized with zero.

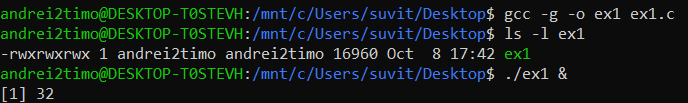
\* .data -- containing global variables that are initialized.

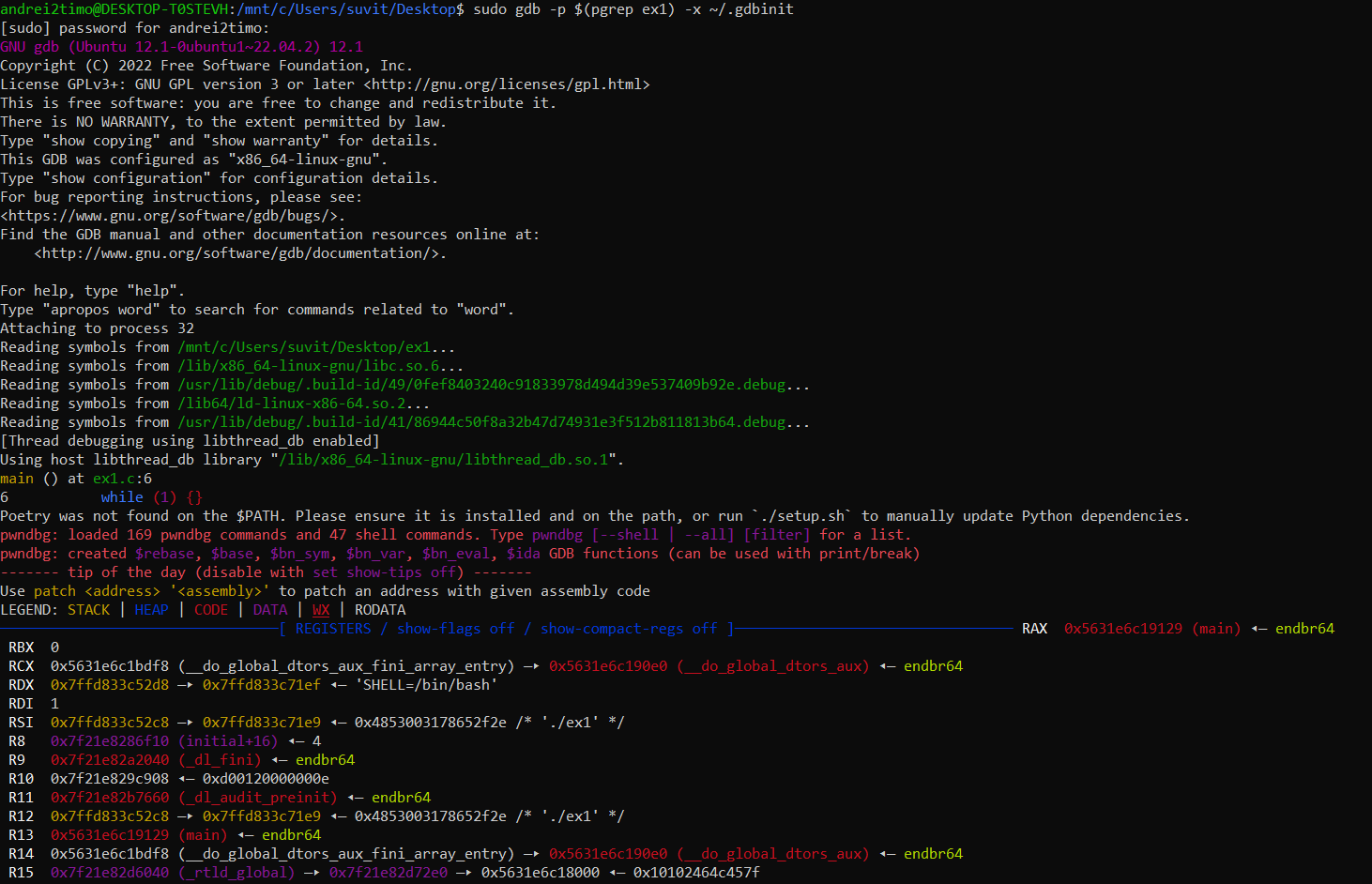
\* .rodata -- containing read-only data, like constants and strings.

\* .text -- containing executable instructions (bytecode).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #include <stdio.h>  char useful[] = "Where is this located?";  int main() {      while (1) {}  } |

Compilare:



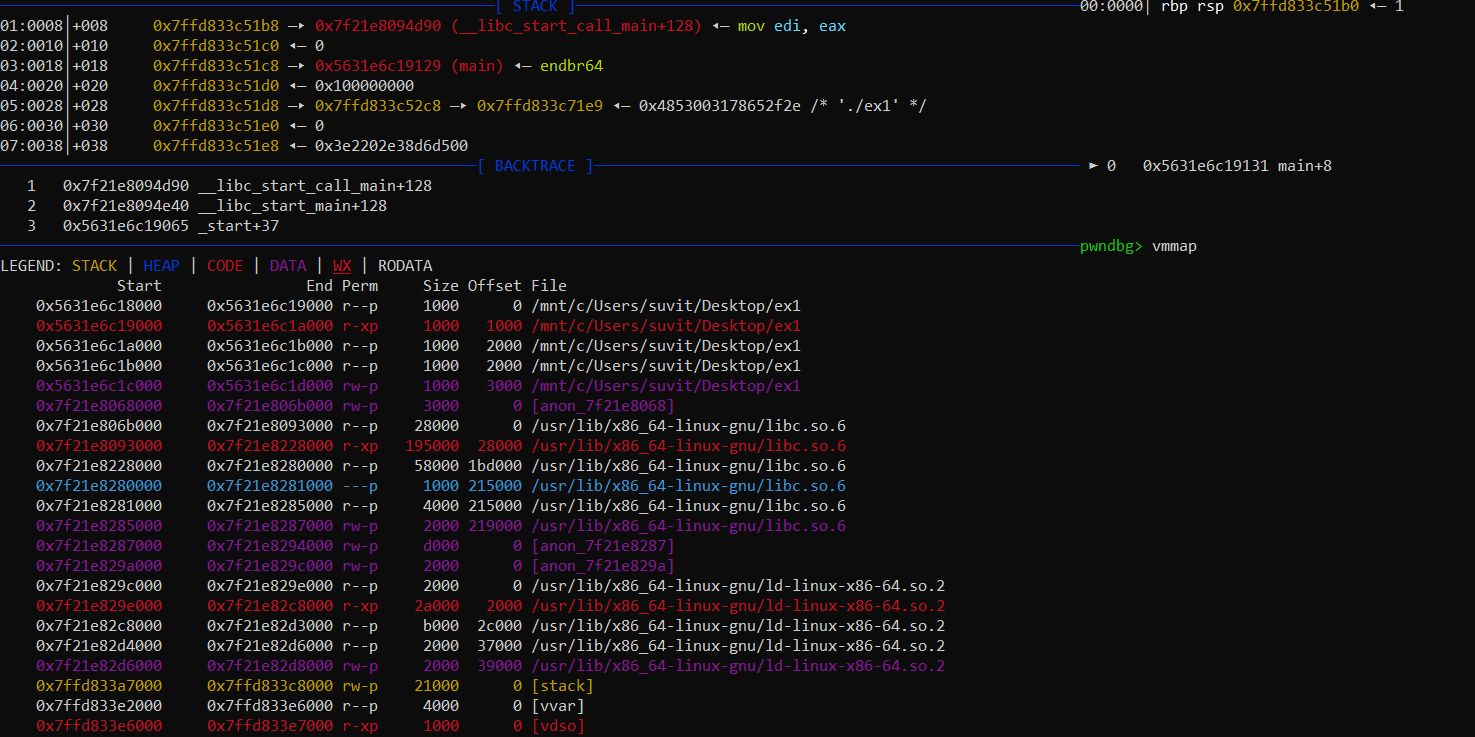
Run **sudo gdb -p $(pgrep ex1) -x ~/.gdbinit** *→ GNU Debugger (GDB) to a running process and execute a specific initialization file.*

Ce reprezintă fiecare informație? → **Size**=dimensiunea secțiunii încărcate; **VMA**=**V**irtual **M**emory **A**ddress; **LMA**=**L**ogical **M**emory **A**ddress; **File off**=offset-ul secțiunii de la începutul fișierului; **Algn**=aliniere; **CONTENTS, ALLOC, LOAD, READONLY, DATA**=flags (dacă o secțiune este LOADED sau READONLY sau etc...). ([Sursă](https://www.thegeekstuff.com/2012/09/objdump-examples/))

Zona de Stack (.eh\_frame) și cea de Heap (.dynamic) nu pot fi identificate (sunt create dinamic în timpul rulării) deoarece programul nostru nu conține *stack frames* (care apar atunci când avem apeluri la funcții) și nici *alocări dinamice* (zona heap e dedicată pentru această problemă). (Sursă: Materialul de Laborator)

Zonele Identificate:

* Text/Code Zone: 0000000000001060
* .data: 0000000000004000
* .bss: 0000000000004010



 **Registers State**:

* The register states give insight into what data and memory locations the CPU is currently working with.
* For instance:
  + RAX points to the main function.
  + RSP (Stack Pointer) and RBP (Base Pointer) are both at 0x7ffd833c51b0, which suggests a typical function call stack frame.
  + The rest of the registers (RBX, RCX, etc.) show their respective values, indicating the program's current operational context.

 **Disassembly**:

* The disassembly section indicates that the main function executes a jump instruction (jmp main+8), effectively creating an infinite loop since it jumps back to itself.
* This is likely a result of the while (1) {} loop in the source code.

 **Stack Memory Layout**:

* The stack layout shows the values at various offsets. The first few lines of the stack (rbp and rsp values) are consistent with the typical stack structure for a function call.
* The stack includes references to \_\_libc\_start\_call\_main, which is part of the C runtime initialization.

 **Backtrace**:

* The backtrace shows the call chain leading to the current execution point. The function main is called, and control has not yet returned to any higher-level function, indicating that the program is still running.

 **Memory Map (vmmap)**:

* This section outlines the virtual memory layout of the process. Notable segments include:
  + **Code Segment**: Contains the compiled code for your program.
  + **Data Segment**: Holds initialized data, such as your char useful[] variable.

From the output provided above:

* The program is currently stuck in an infinite loop due to the while (1) {} statement, which is common for debugging scenarios where you want to inspect the state but can be problematic in practice.
* The code and data segments have been properly allocated, and the debugger is able to show the current state of the registers and memory addresses effectively.

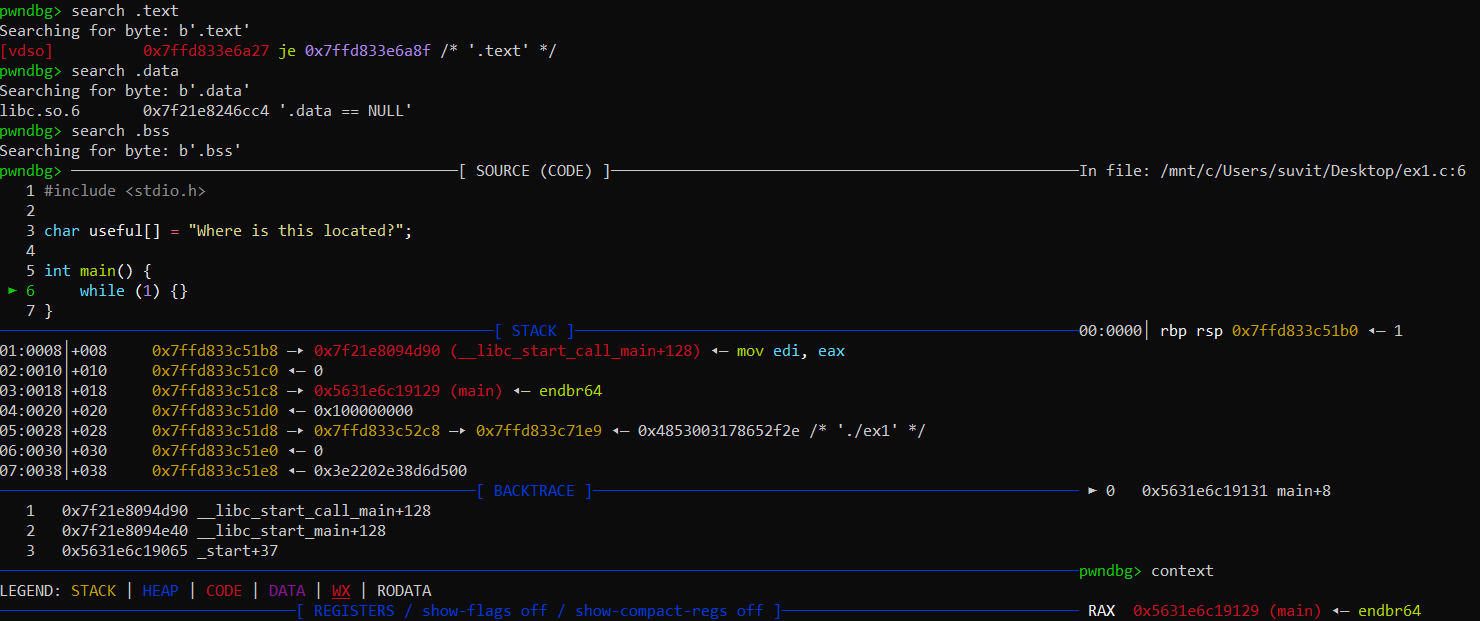
Commands:

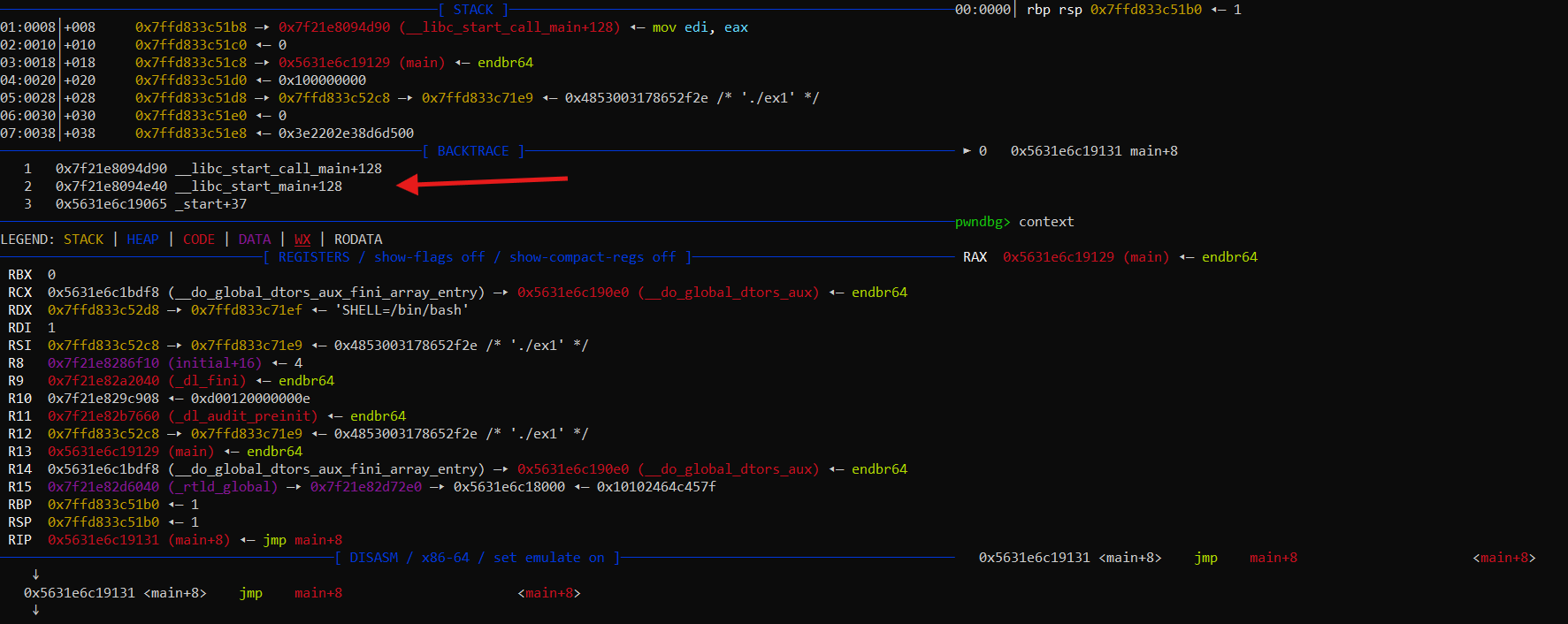
* cd /mnt/c/Users/andrei2timo/Desktop
* gcc -g -o ex1 ex1.c
* ./ex1 &

In another terminal:

* sudo gdb -p $(pgrep ex1) -x ~/.gdbinit
* pwndbg> vmmap
* pwndb> context

**\*\*[Q1]\*\*: Where is each section mapped? Try using the `search` command in `pwndbg` (or `search-pattern` in `GEF`).**

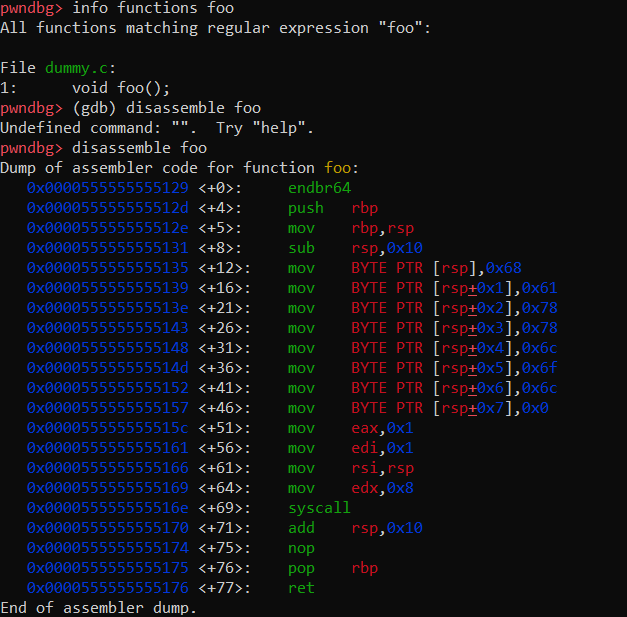
****

****

**\*\*[Q2]\*\*: Try finding the address of `foo()` in gdb and printing its disassembly.**

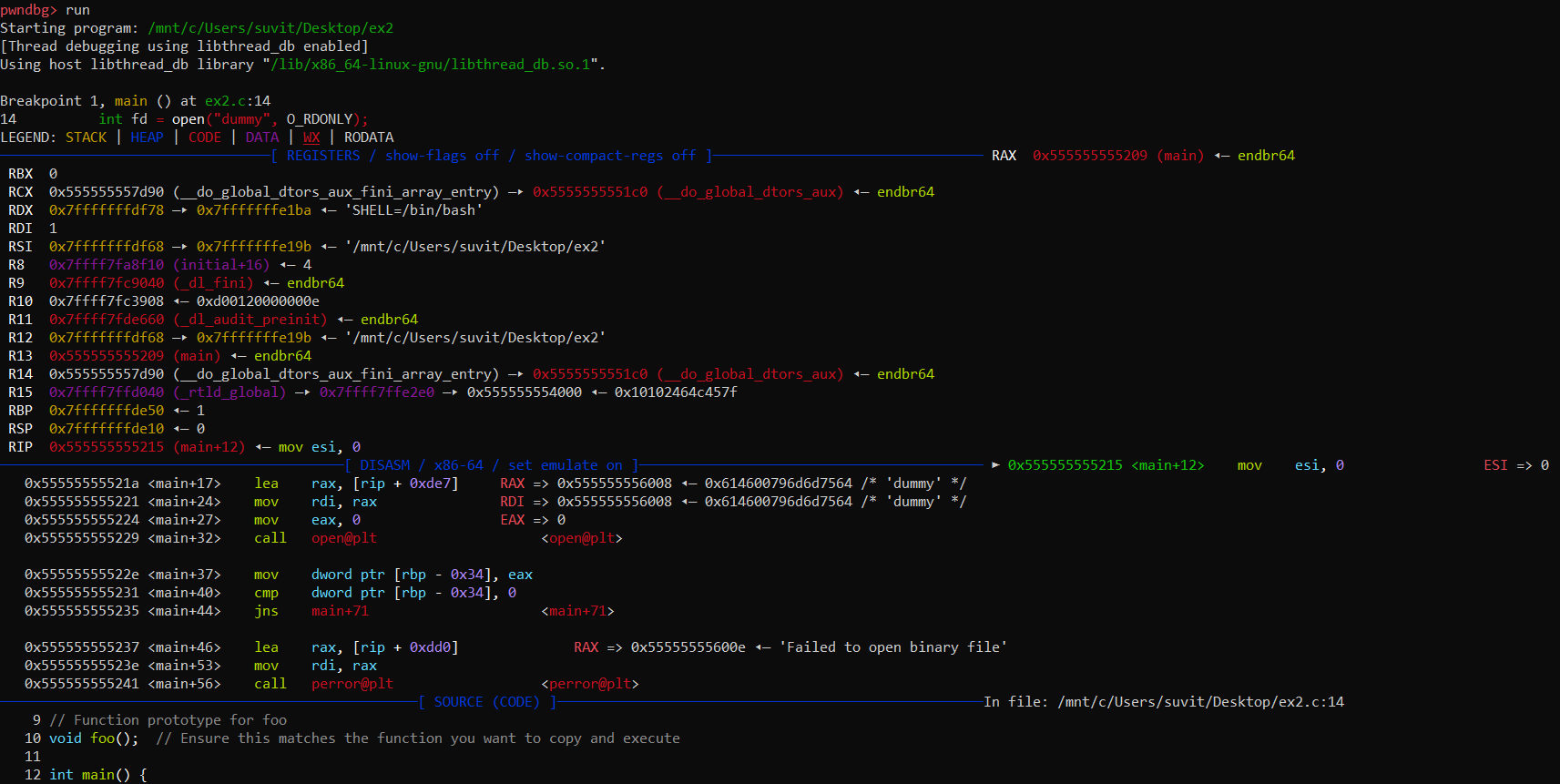
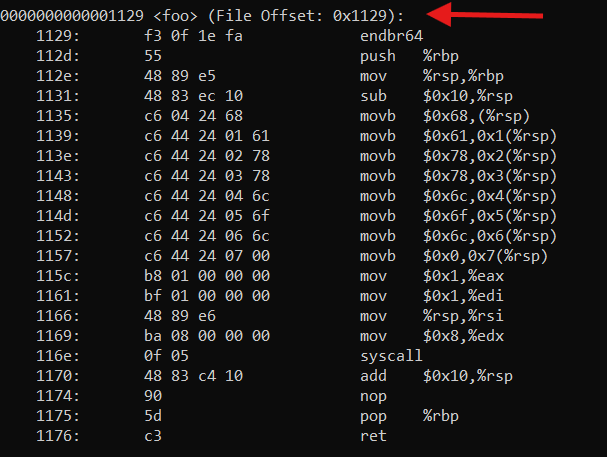
gcc -g -o dummy dummy.c

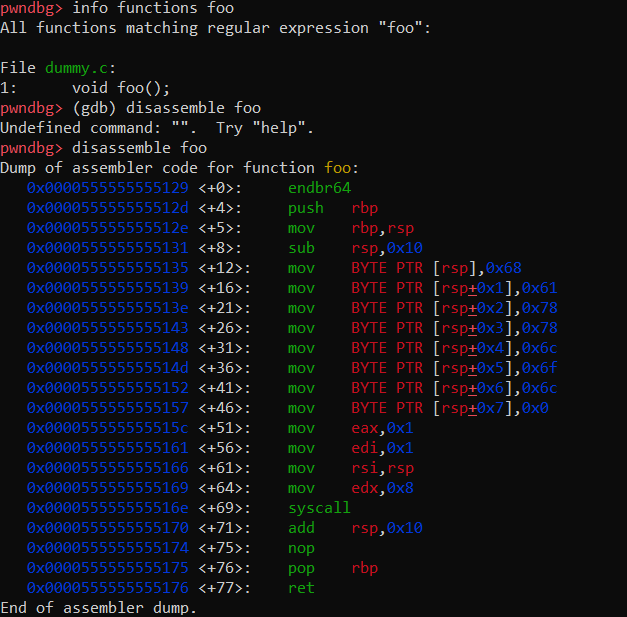
gdb ./dummy

****

**Ex 2**. Quick reminder: When we are talking about executable instructions, we are talking about "bytecode" -- a sequence of bytes that the CPU reads and executes. All executable code on the computer gets at some point or another translated to this bytecode, as it's the only thing the CPU understands. Essentially, all high-level languages need to be translated to this. Usually, the \*compilation process\* does this, going from high level language (C) --(to)--> low level language (assembly) --(to)--> bytecode. For the course and lab, we are mostly going to use the x86\\_64 architecture as a backbone for everything we do. If you need to know anything about its assembly language, I recommend the [Intel Software Developer's Manual](https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html) and a site for looking up instruction mnemonics easier, like [this one](https://www.felixcloutier.com/x86/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | #include <stdio.h>  #include <stdlib.h>  #include <fcntl.h>  #include <sys/mman.h>  #include <unistd.h>  // Define the function pointer type for the function we are loading  typedef void (\*foo\_t)();  int main() {      // Open the binary file      int fd = open("dummy", O\_RDONLY);      if (fd < 0) {          perror("open");          return EXIT\_FAILURE;      }      // Get the size of the file      off\_t size = lseek(fd, 0, SEEK\_END);      lseek(fd, 0, SEEK\_SET);      // Allocate a memory region with mmap      void \*mem = mmap(NULL, size, PROT\_READ | PROT\_WRITE | PROT\_EXEC, MAP\_PRIVATE, fd, 0);      if (mem == MAP\_FAILED) {          perror("mmap");          close(fd);          return EXIT\_FAILURE;      }      // Now we need to find the offset for foo()      size\_t foo\_offset = 0x1129;  // Make sure this offset is correct for foo()      // Create a function pointer to the loaded code      foo\_t foo = (foo\_t)((char \*)mem + foo\_offset);      // Execute the loaded function      printf("Calling foo()...\n");      foo(); // Call the function      // Clean up      if (munmap(mem, size) == -1) {          perror("munmap");      }      close(fd);      return EXIT\_SUCCESS;  } |



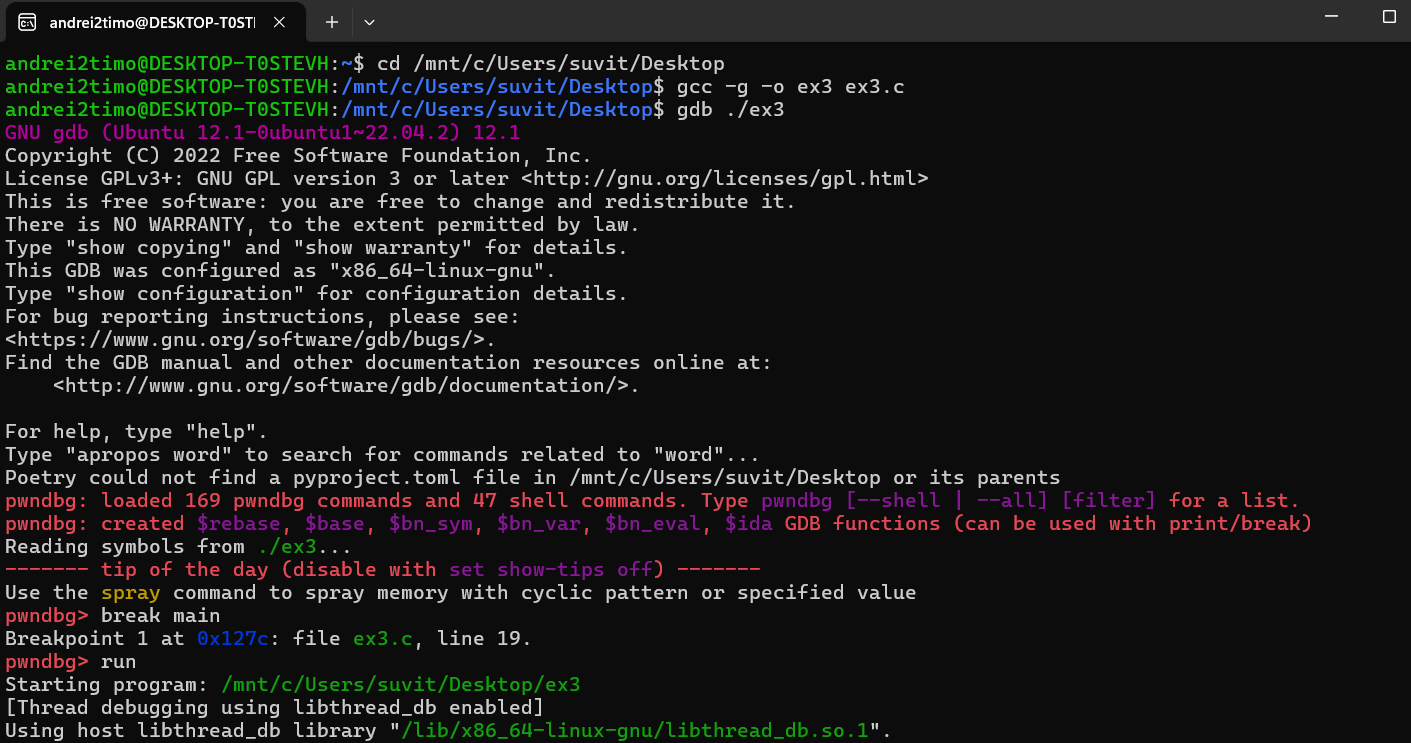


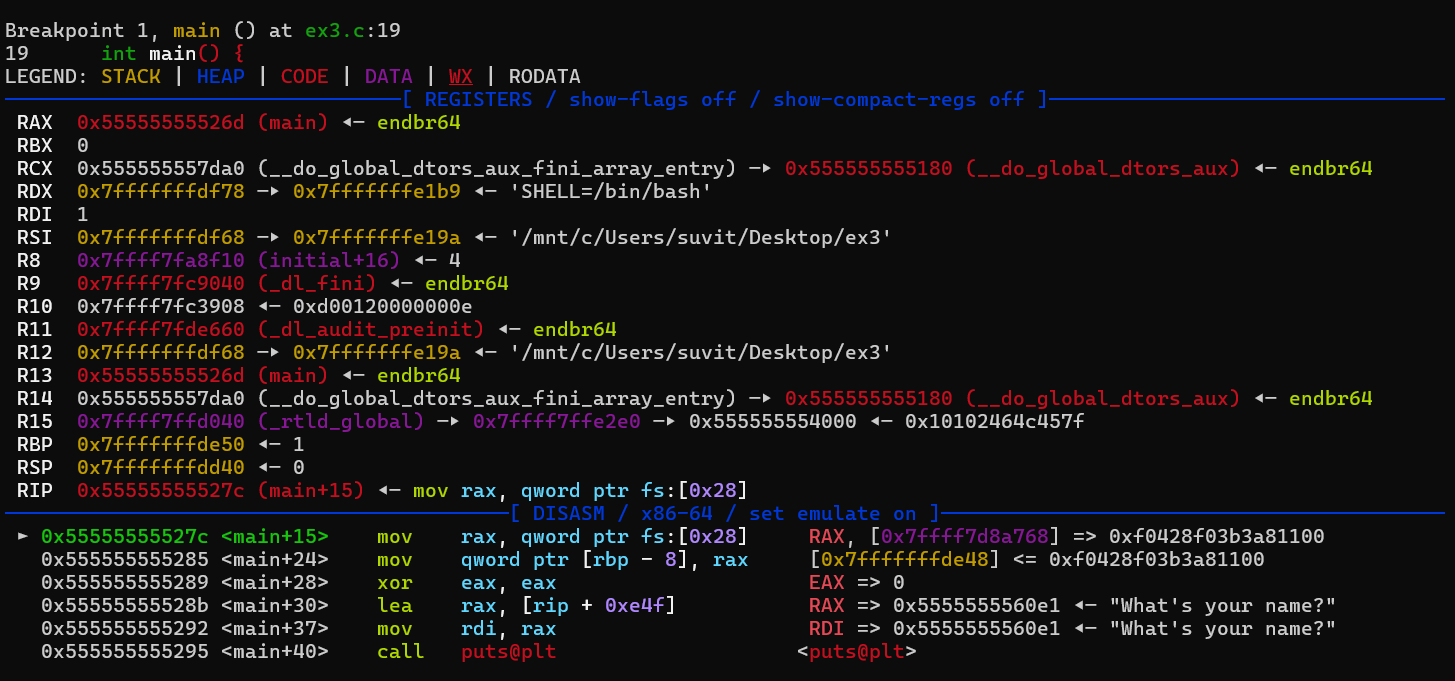
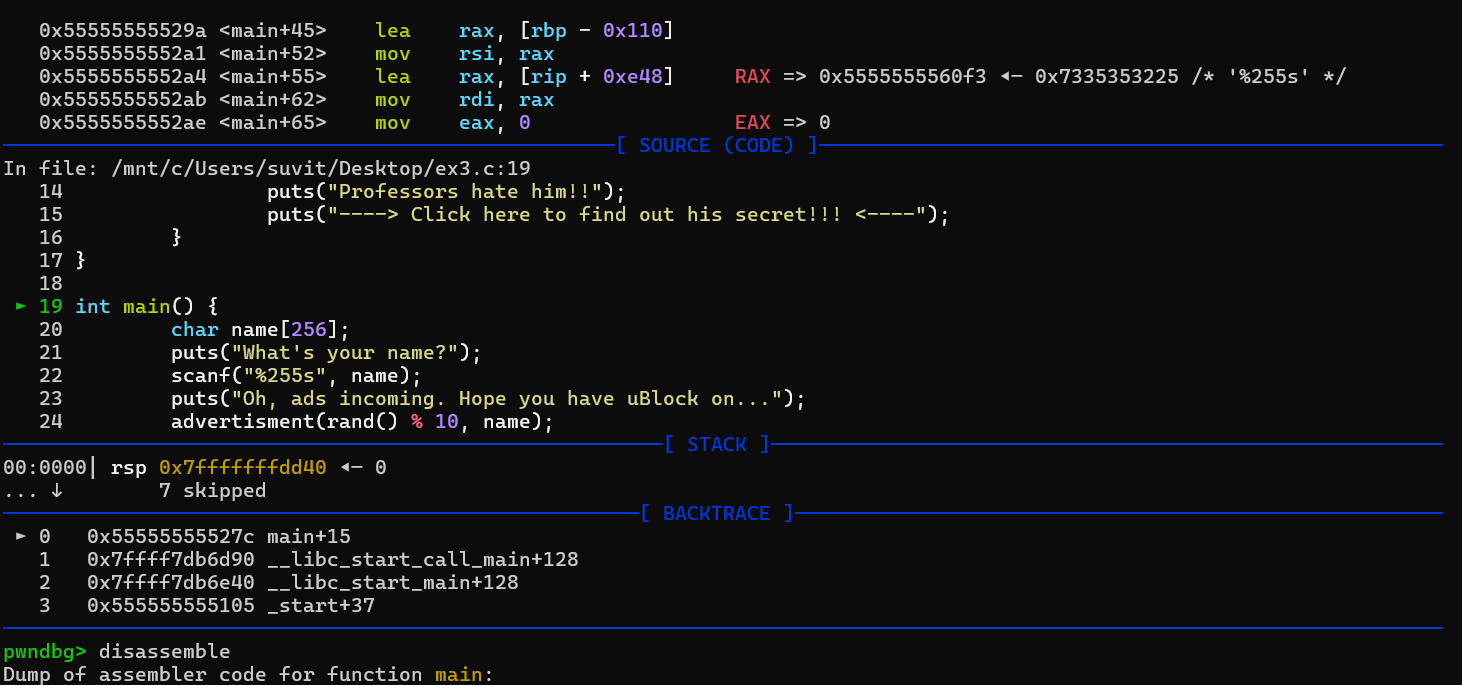
\*\*[Q3]\*\*: Check `gdb` with your binary. How does `vmmap` look after running `mmap`? You can step through each line of code with `next` or `n`. You can step through each assembly instruction with `next instruction` or `ni`.

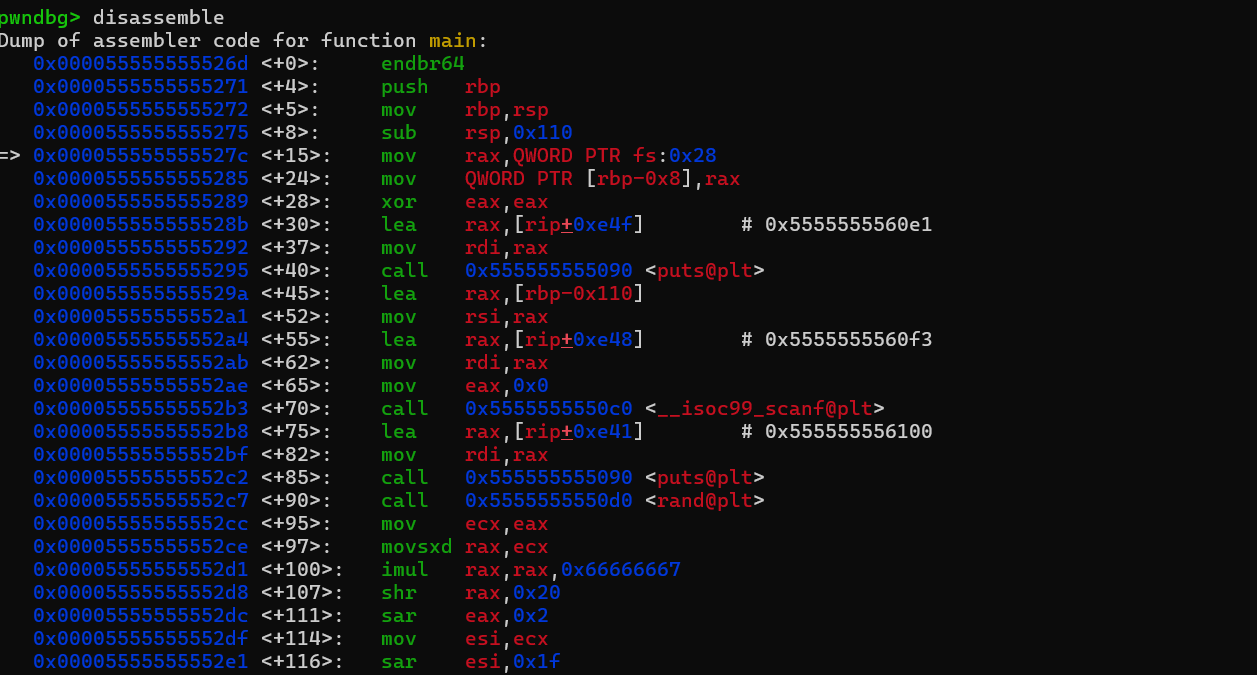
* 1. Compile the Program with Debugging Information: We start by compiling the program (ex2.c) with debugging symbols. This enables gdb to provide detailed information about the source code during debugging. The -g flag is crucial as it includes the debugging information in the executable.
  2. Launch gdb: We open gdb with our compiled binary (ex2). This initializes the debugging environment.
  3. Set Breakpoints: To examine the program’s execution flow, we set a breakpoint at the start of the main function. This allows us to pause execution before the program performs any significant actions, such as memory allocation.
  4. Run the Program: We then run the program within gdb. The execution will halt at the previously set breakpoint, allowing us to inspect the state of the program.
  5. Step Through the Code: Using the next command, we step through each line of the main function until we reach the mmap call. This process helps us understand the flow of control and when memory is allocated.
  6. Check Memory Mapping: After the mmap call is executed, we use the info proc mappings command to inspect the memory mapping of the process. This command provides a detailed list of the memory regions allocated to the process.

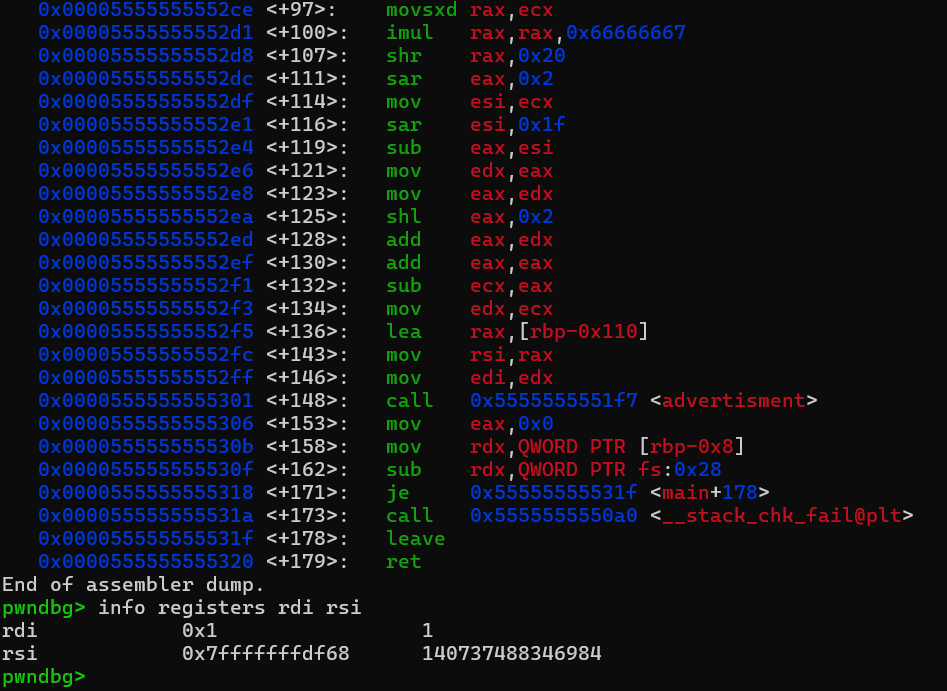
**Ex 3**.

\*\*[Q4]\*\*: Can you identify the arguments of a function call in the disassembly?









**Step 1:** Compile the ex3.c program with debugging symbols

***cd /mnt/c/Users/suvit/Desktop***

***gcc -g -o ex3 ex3.c***

This will generate the ex3 executable with debugging information.

**Step 2:** Start GDB

Open GDB and load the ex3 executable:

*gdb* ***./ex3***

This will load the program into the debugger.

**Step 3:** Set a breakpoint in main

Set a breakpoint in the main function so that you can inspect the disassembly before and during the function call.

**gdb break main**

**Step 4:** Run the program

Run the program to hit the breakpoint:

***gdb run***

**Step 5:** Disassemble the `main` function

Now, let's inspect the disassembly of the main function. This will display the assembly instructions generated for main, including the function call to advertisment.

***gdb disassemble***

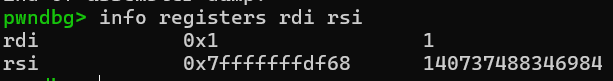
This will display the assembly code for the main function. You will see instructions corresponding to loading the arguments into the appropriate registers before calling advertisment (see pictures above).

**Step 6:** Identify the arguments for the advertisment function

- The first argument (num) is passed in the `rdi` register.

- The second argument (target) is passed in the `rsi` register.

You should see something like this in the disassembly:



- The value in rdi is the number of ads (`num`).

- The value in rsi is the pointer to the string (the user's name).

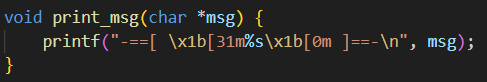
- The function call follows the x86\_64 calling convention, which passes the first argument in rdi and the second in rsi.

By analyzing these registers in the disassembly, you have identified the function arguments for the call to advertisment.

\*\*[Q5]\*\*: Did you get a `SIGSEGV` in `printf()`? What causes it? `pwndbg` hints at the reason.

Yes, I encountered a SIGSEGV (Segmentation Fault) in printf() when attempting to manipulate function arguments using GDB (pwndbg).

A segmentation fault (SIGSEGV) occurs when the program tries to access memory it is not allowed to. This typically happens due to an invalid pointer dereference.

In this case, the issue happened during the execution of the printf() function in the print\_msg() function:

Here, printf() expects a valid pointer (msg) as the first argument, stored in the rdi register according to the x86-64 calling convention.

When debugging with pwndbg, I modified the rdi register, which holds the first argument (msg), and accidentally set it to 0x1. This is an invalid memory address, as 0x1 does not point to any valid string data in memory.

When printf() tried to dereference this address and treat it as a pointer to a string, it resulted in a segmentation fault because the program attempted to access memory it didn’t have permission to access.

**Root Cause:**

* The register rdi (which holds the first argument msg for print\_msg()) was set to an invalid address (0x1).
* printf() expected a valid pointer to a string, but instead received an invalid memory address, leading to the SIGSEGV.

**Key Point:**

In x86-64 calling conventions, function arguments are passed via registers:

* The first argument goes into rdi.
* If this register holds an invalid memory address (as it did in this case), any function expecting a valid pointer will likely crash with a segmentation fault.