TITLE

How to CTF: A Crash Course in Reverse Engineering

REQUIRED EQUIPMENT/SOFTWARE

1. Linux OS
2. GDB (peda recommended - https://github.com/longld/peda)
3. Objdump
4. Python
5. If running a 64-bit OS, ensure it can run 32-bit programs (may have to run "sudo apt-get install lib32ncurses5 lib32z1")

ORDER OF OPERATIONS

1. What is a CTF?
2. Introduction to Assembly
3. Memory and the Stack
4. Registers
5. Creating a New Stack Frame
6. Function Calls
7. Example - Reversing
8. Example - Exploitation

WHAT IS A CTF?

- ➢ A type of information security competition.
- ➢ Comes in three common formats: Jeopardy, Attack-Defense, and Mixed.
- ➢ In all formats, the main object is to procure various "flags" which are then submitted for points.
- ➢ Jeopardy style CTFs contain a range of categories to include Web, Forensics, Crypto, Reversing, etc. Solving a problem produces a flag which is worth a varying amount of points based on the difficulty of the problem.
- ➢ The goal of reversing is to understand the functionality of the given program in order to exploit the code and gain the flag

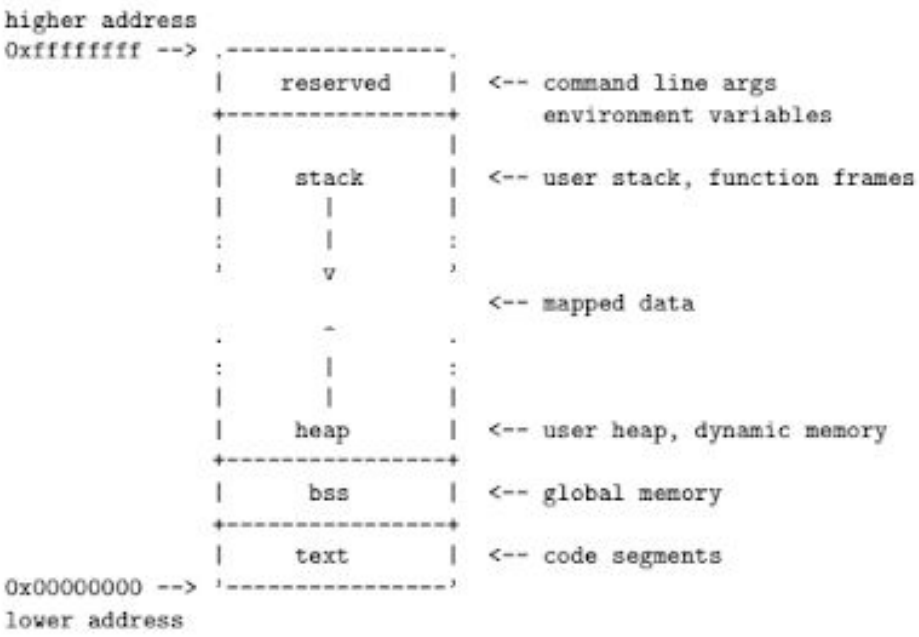
INTRODUCTION TO ASSEMBLY

- ➢ A low level language that is a step above machine code.

- ➢ All programs are converted to assembly before they run which is either done by the user (ex. Using gcc to compile C) or by the program (ex. Running javac to compile a java program to java byte code).
- ➢ The compiler turns written code into op codes which are viewed as hexadecimal numbers which directly correspond to assembly instructions.
- ➢ We can view the assembly of an executable even without the source code.

MEMORY AND THE STACK

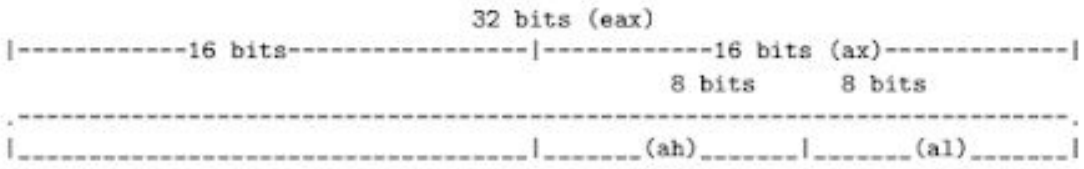
- ➢ In order to properly reverse engineer a program, we first have to understand how memory is utilized when we run a program.
- ➢ The typical layout of a simple computer's program memory is shown below.



- ➢ In memory, functions store their temporary information in the stack (including the main() function).
- ➢ A new stack frame is allocated at the beginning of each function call for that function's use and after the function ends, that space in memory is deallocated.
- ➢ The stack is very important in reversing and exploiting binaries.

REGISTERS

- ➢ All data is stored in either memory or in a register.



- In a 32 bit operating system every register is 32 bits.
- A register is like a variable, except that there are a fixed number of registers.
- A register is the only place where math can be done (addition, subtraction, etc).
- Movement of values between registers and memory is very common.
- Intel assembly has 8 general purpose 32-bit registers: eax, ebx, ecx, edx, esi, edi, ebp, esp.
    - eax: to store the return value of a function and as a special register for certain calculations, such as multiplication and division.
    - ebx: has no specific uses.
    - ecx: occasionally used as a function parameter or as a loop counter.
    - edx: generally used for storing short-term variables within a function.
    - esi: often used as a pointer.
    - edi: often used as a pointer.
    - ebp: also known as the base pointer. Points to the base of the current function's stack frame.
    - esp: also known as the stack pointer. Points to the current position of the current function's stack frame.
- Special purpose registers: eip, eflags
    - eip: also known as the instruction pointer. It tells the computer where to go next to execute the next command and controls the flow of a program.
    - eflags: there are 17 necessary flags and each flag holds one to two bits. Most commonly used when the computer compares two registers.

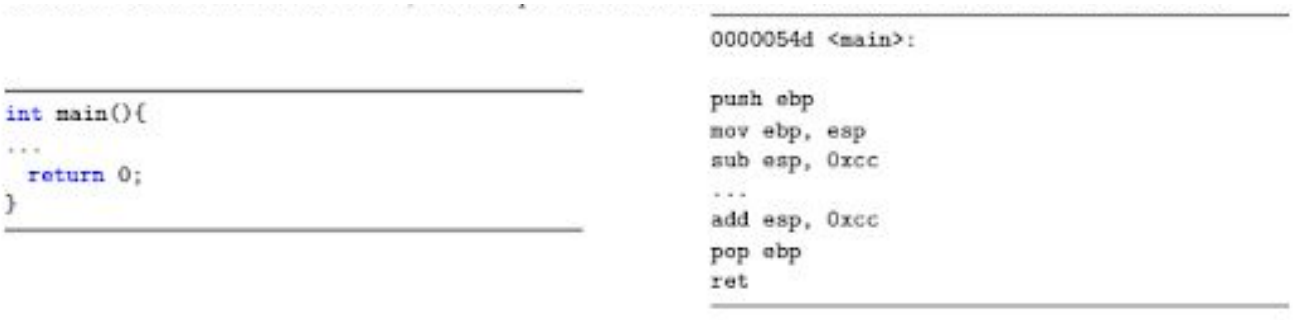
BASIC INSTRUCTIONS

- Two major types of syntax: Intel and AT&T.
- We will be using Intel but it is important to understand both.
- Format for instructions that take no operands: **INSTR**
- Format for instructions that take 1 operand: **INSTR** arg
- Format for instructions that take 2 operands: **INSTR** dest, src (for Intel syntax)
- https://en.wikibooks.org/wiki/X86_Assembly/X86_Instructions
- When reading through a decompiled binary, you do not want to get stuck trying to read and understand every single line of code.

- The 2 main instructions to look out for when reversing assembly are: jmp (with variants) and call. To do anything major, a program will call a system function such as printf, gets, or strncmp. Jmp gives you a feel for the control flow of the program.

CREATING A NEW STACK FRAME

- Every function has its own frame on the stack, so whenever a new function is called, the computer has to create a new stack frame for the new function.

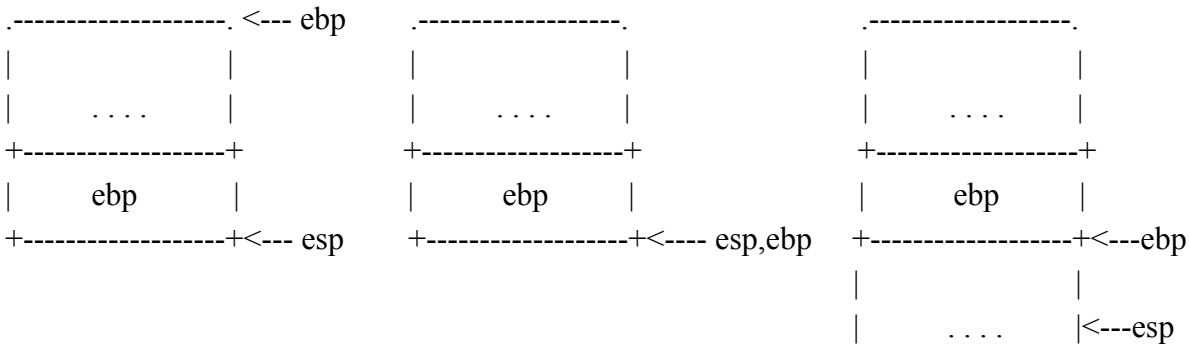


```
int main(){
...
  return 0;
}
```

```
0000054d <main>:

push ebp
mov ebp, esp
sub esp, 0xcc
...
add esp, 0xcc
pop ebp
ret
```

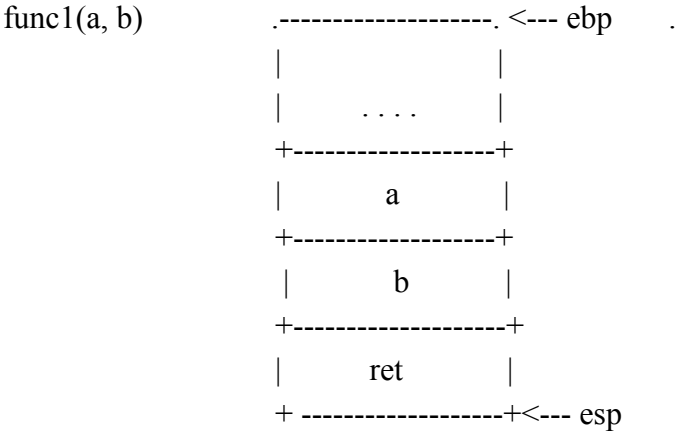
- The above graphic depicts C code on the left with the resulting assembly code on the right.
- The stack frame is bounded by ebp and esp, therefore, to go into a new function, you have to move ebp and esp to a new area in memory.
- Ebp is pushed onto the stack to preserve the location of the previous stack frame for when we return from the new function call.
- Then ebp is moved down to where esp is on the stack (mov ebp, esp puts the value of esp into ebp) and finally a value is subtracted from esp which which moves esp down the stack to create the frame for the function.
- The amount that esp is subtracted from is dependent on how much space the function needs for variables.



- At the end of a function, the program has to do the opposite of what it did to start with.

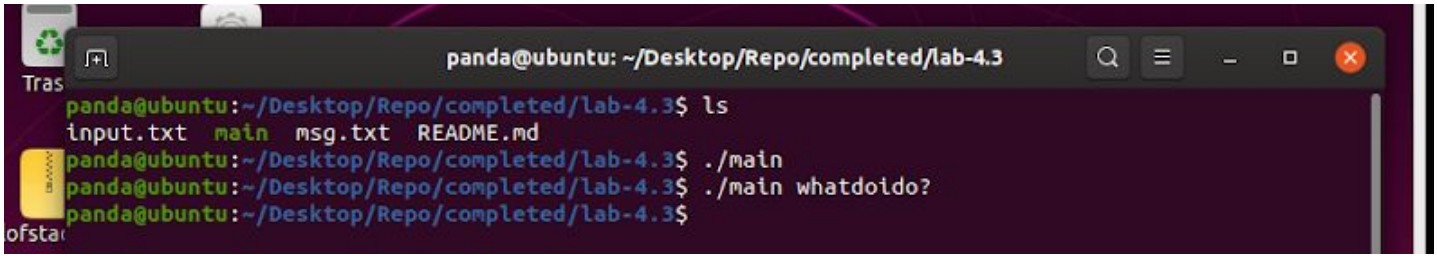
FUNCTION CALLS

- ➢ Before a function is called, the stack has to be setup to contain any arguments that the function takes.
- ➢ Arguments are pushed onto the stack in reverse order.
- ➢ Lastly, before the new stack frame is setup, the return address is pushed onto the stack so that the program knows where to continue the control flow at the end of a subroutine.
- ➢ After the subroutine is called, the function arguments are accessed by calling “ebp+0x8” or “ebp+0xc”. Ebp is used as the reference point because, unlike esp, ebp does not change once the stack frame has been established. (Note: hex numbers reference the number of bytes therefore 0x4 means 32-bits.)

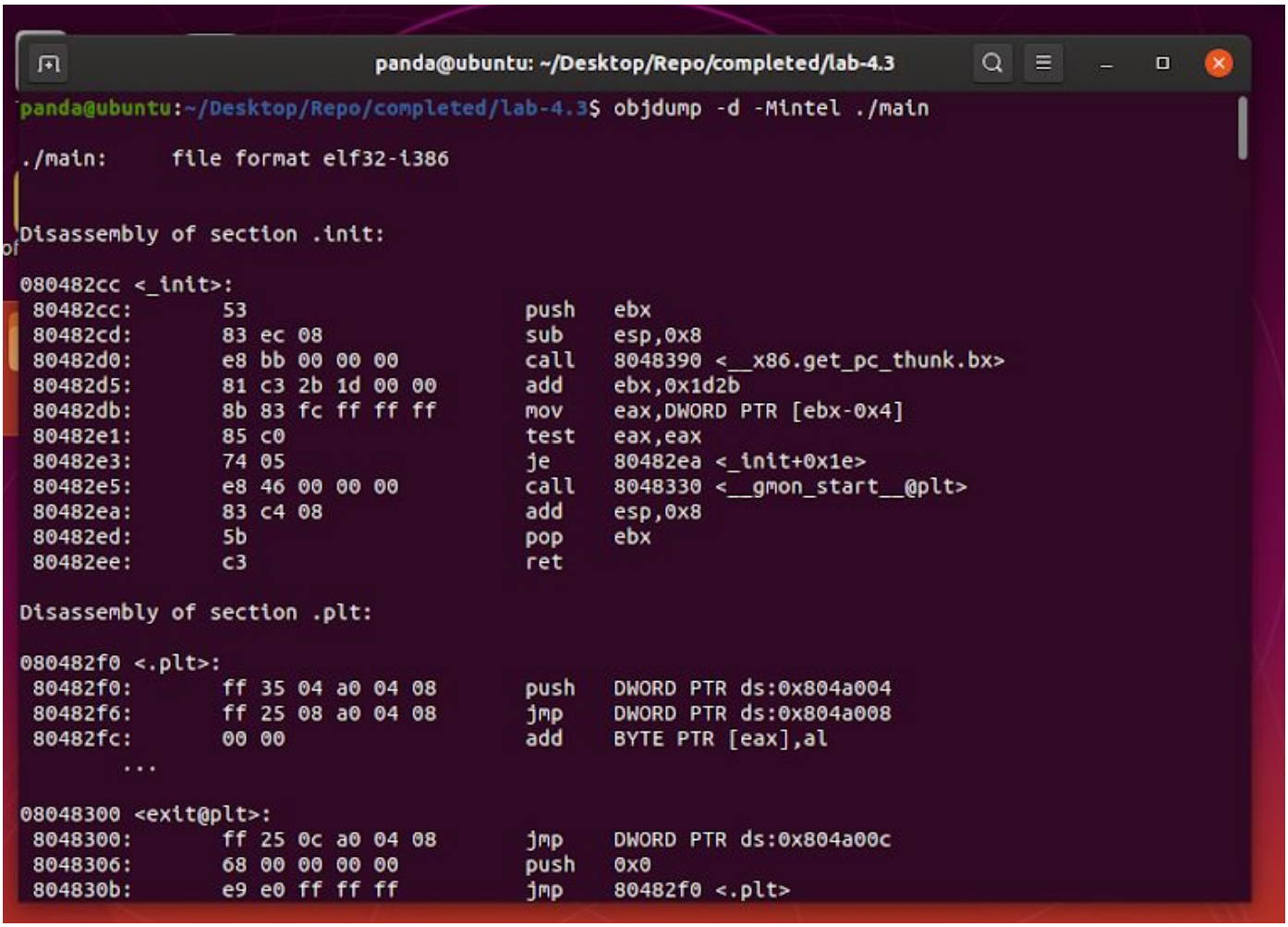


EXAMPLE - REVERSING (OBJDUMP)

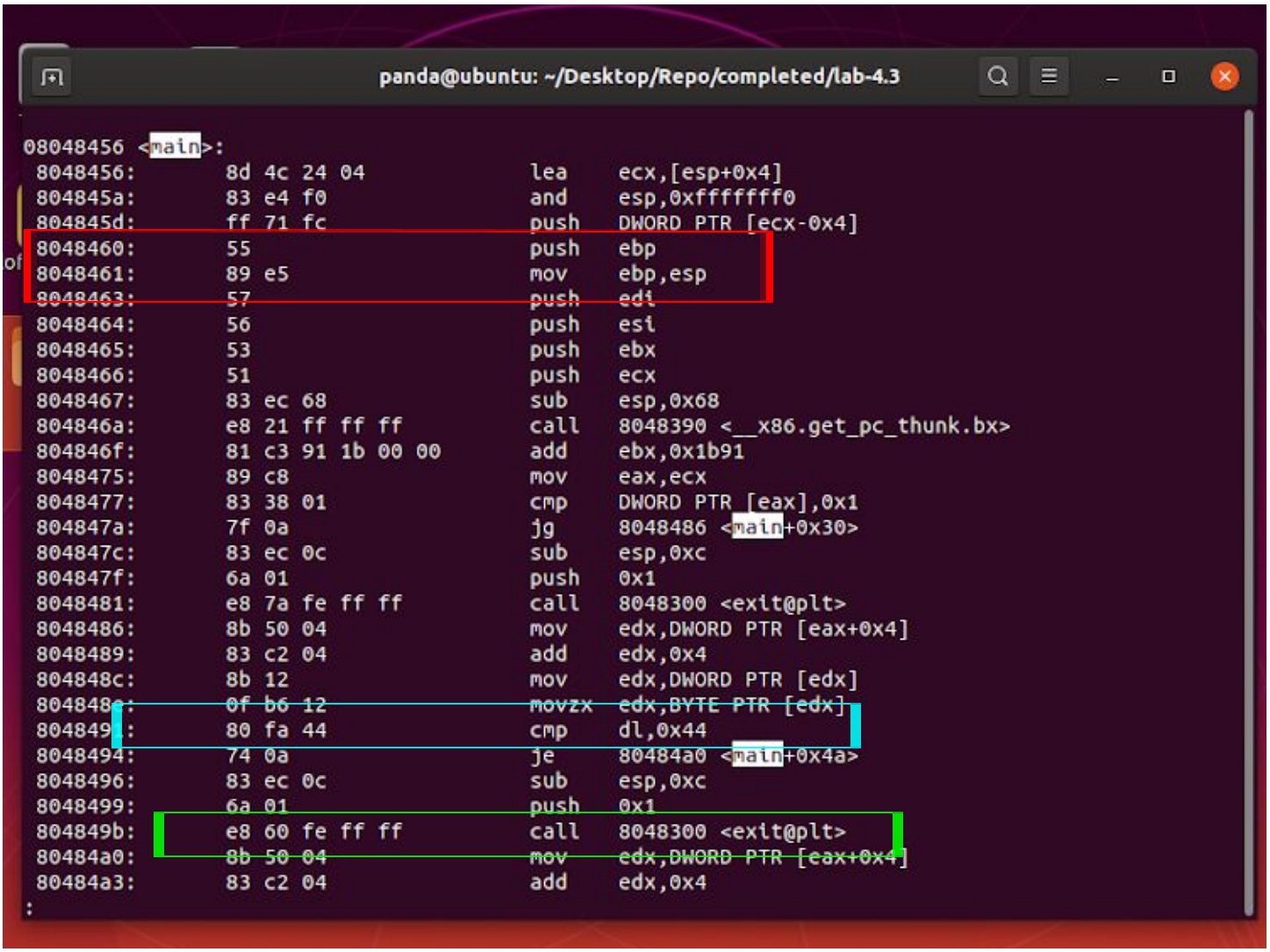
- ➢ When running the first example problem, we can see that it produces no output



- ➢ Since we cannot produce any noticeable output from the executable, our next step would be to analyze the assembly.
- ➢ A command you can utilize to disassemble (translates machine code to assembly) an executable is “objdump”.



- ➢ The command we will run in this example is “objdump -d -Mintel ./main”. This will ensure that the disassembled code is in Intel syntax. The result of running “objdump” will be all the assembly instructions executed when running the program.
- ➢ Since the first function that any executable runs is main(), we will start our investigation there.
- ➢ **You can utilize the command “less” in order to produce a more readable friendly format: “objdump -d -Mintel ./main | less”**



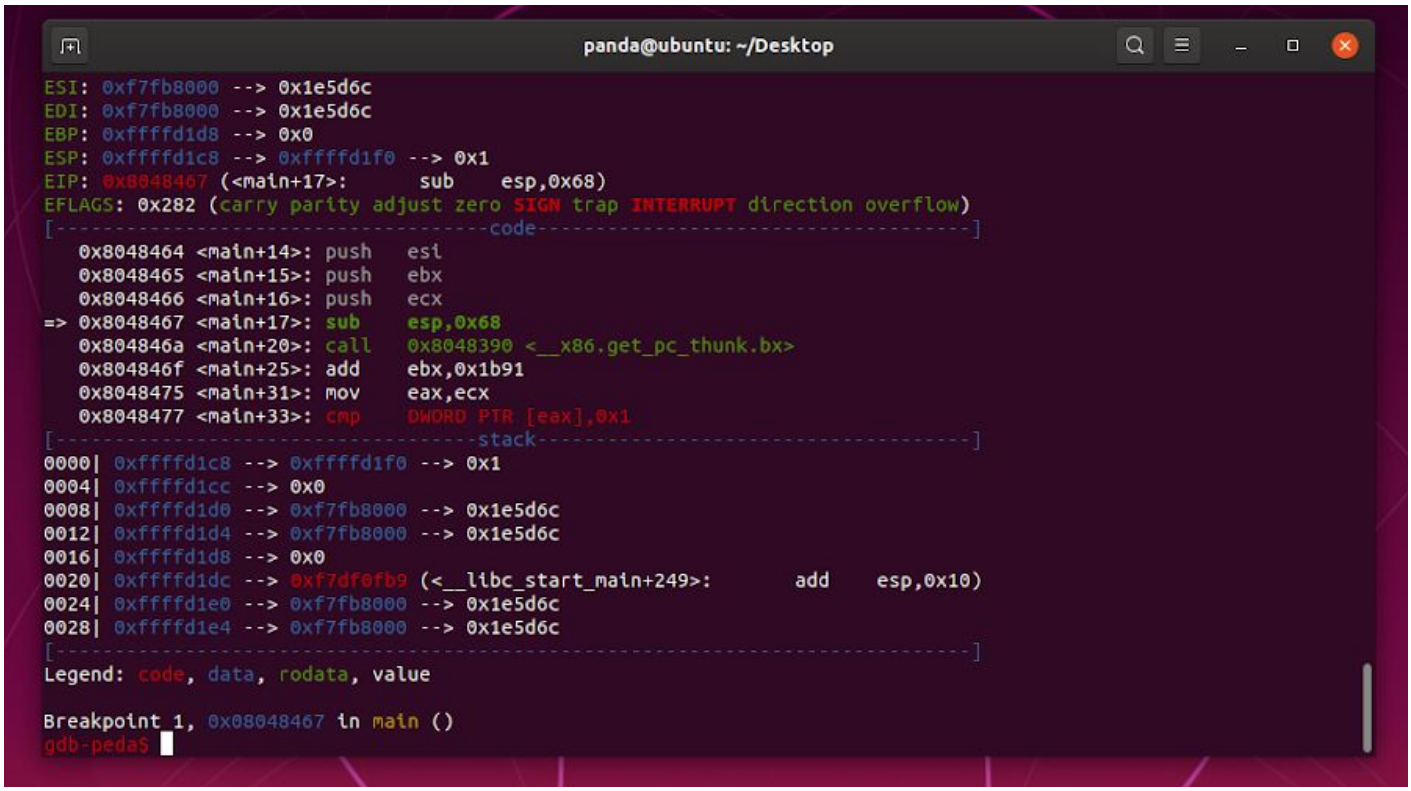
- As you can see in the red box, the main function is setting up its stack frame.
- As mentioned earlier, a key instruction to look out for is “call”. In this example, we can see that the program is calling the exit function which will just exit out of the program. This indicates that we have to somehow avoid this instruction from ever being executed.
- If you look just above the call instruction is a compare instruction. Compares are done to test for a condition, and if the condition is met, it alters the flow of the program.
- The register dl is being compared to 0x44. This value is key, because any good reverser knows that this value is in the ascii table. The value 0x44 corresponds to the letter “D”.
- The program is testing to see if the value in the dl register is 0x44 or “D”. If it is, then the program jumps over the exit instruction.
- If you continue this methodology through the entire program, you will arrive at the answer.

- **Answer**: Dijkstra

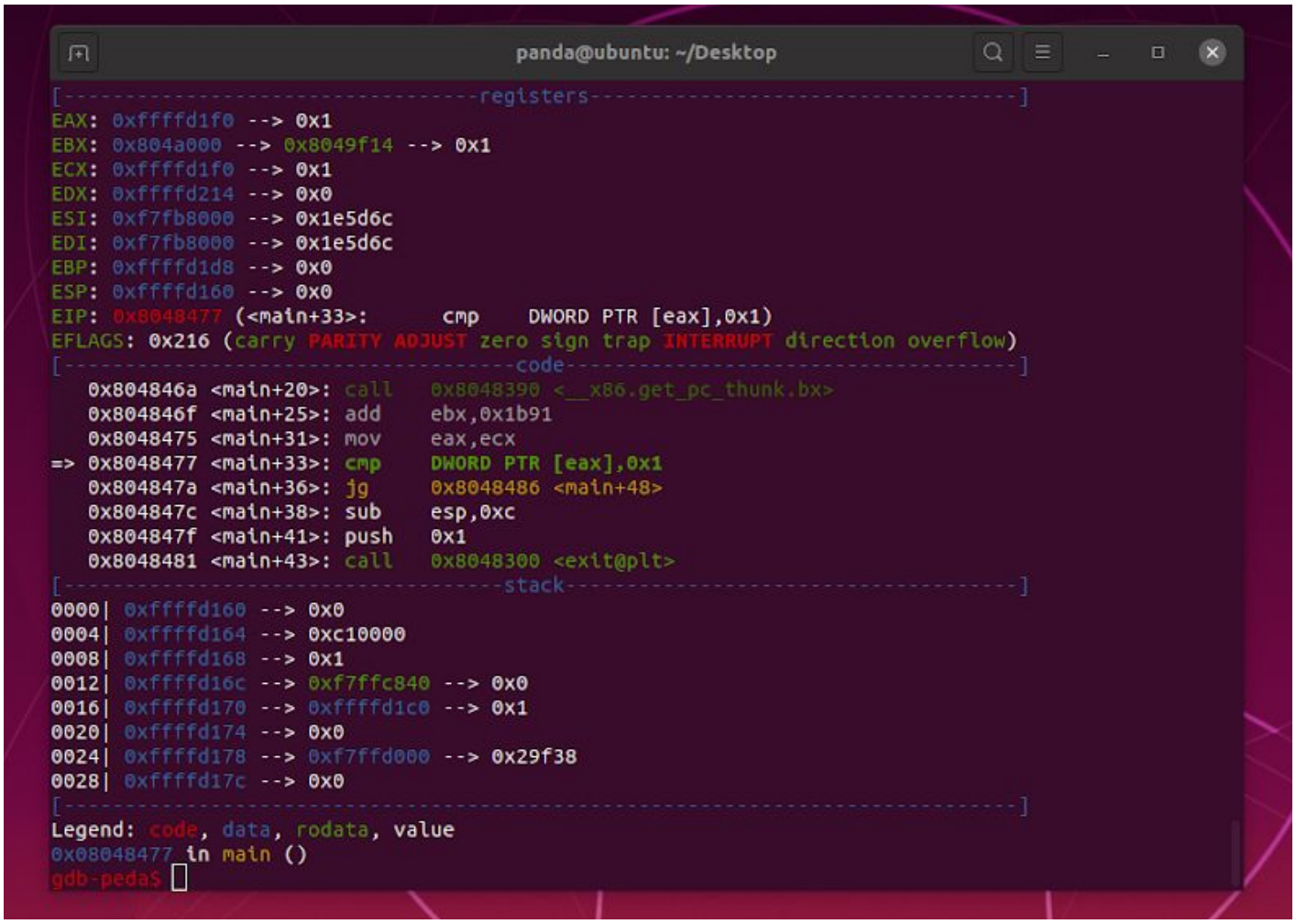
EXAMPLE - REVERSING (GDB)

- GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes
- GDB allows you to step through and alter a program unlike objdump
- GDB cheat sheet: https://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

- We start by setting a breakpoint at main. A breakpoint tells GDB to stop the program at that particular part. We can set a breakpoint at a specific function (b <func name>) or a specific memory address (b *address).
- You can also disassemble in GDB by using the command “disass <funct nam>”.



- ➢ Using the command “run” will start normal program execution until it hits the first breakpoint.
- ➢ The command “step” and “next” will let you go to the next instruction with the difference being “next” will not dive into a subroutine/function.
- ➢ As you step through the program, you can watch the values of the register change

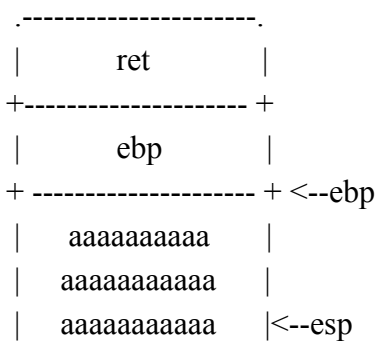


- ➢ After a few steps, we arrive at the first comparison, fail the check, and exit the program. If you haven't figured out what this first check is for, the program is checking to see if the user provided an argument. If not, then the program exits.
- ➢ GDB will also show part of the stack but you can also manually view any part of memory by using "print" or "x". (For more complex reversing, I typically use "x/#xw <location>" because this prints out the contents of memory from the starting location formatted as a word - 4 bytes - for as many times as indicated)
- ➢ Using the GDB cheat sheet, try stepping through the program and testing out different commands.

EXAMPLE - EXPLOITATION

- ➢ Now that we understand assembly and how the stack works, we can start learning how to exploit it.

- When writing a program that requires user interface, what, as coders, do we usually require? User input!! So where does that input get stored?
- What happens if you allow a user to input as much information as they want without verifying the length of the input?
- If my buffer is 0x20 bytes big and a user enters in 0x21 bytes of data, what issue do we run into?



- We can overwrite information that is already on the stack!!! In doing so, we can also manipulate the control flow of a program.
- The return address tells the program where to go once a subroutine is over, so if we change that address, we control what the program executes next.
- Must keep endian-ness into account.
- **Answer**: printf $(python -c "print 'A'*22 + '\xc8\x85\x04\x08'") | ./vuln