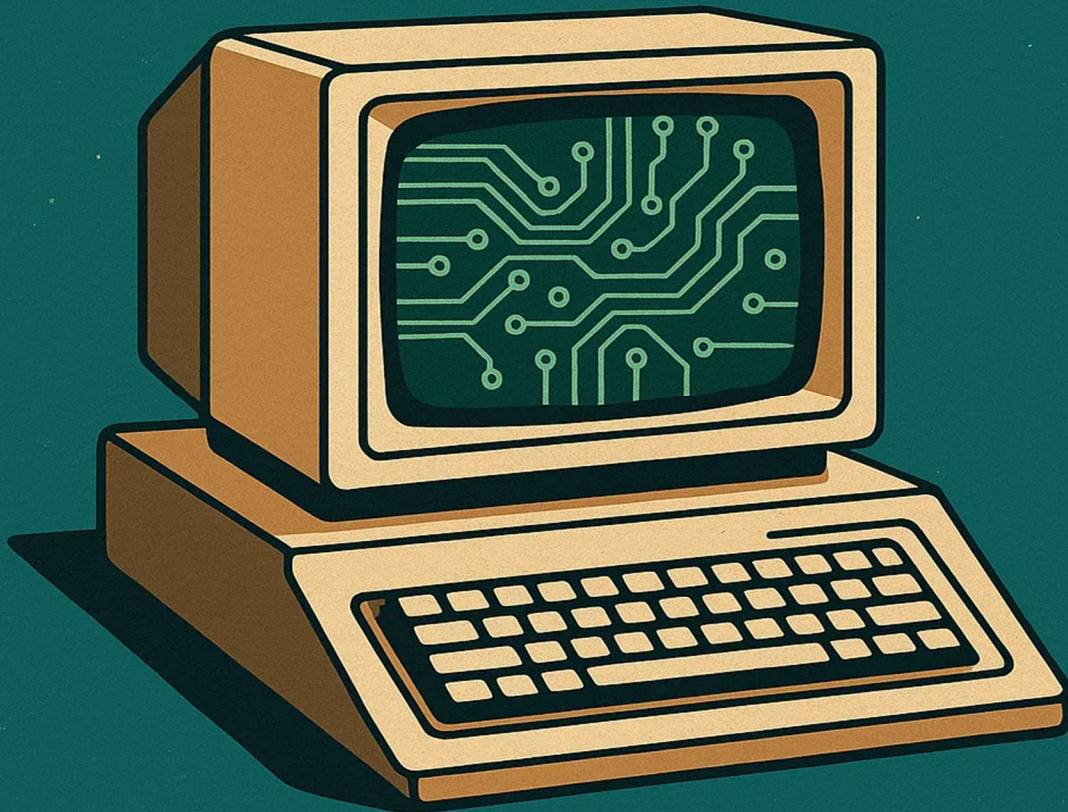


RE:BOOT

— AN INTRODUCTION TO —
**REVERSE
ENGINEERING**



WORKSHOP WORKBOOK

Contents

Introduction.....	3
Legal and Ethical Considerations in Reverse Engineering.....	3
Understanding Computer Architecture	4
Memory and Registers: Stack, Heap, and CPU Internals	5
Software and Hardware Abstractions	7
Instruction Set Architecture (ISA) and Endianness	8
Lab 1: Disassembling Source Code C++ to Assembly	9
Step 1: Write and Save the C++ Program	9
Step 2: Compile and run the area.cpp file	10
Step 3: Compile and add the -S flag to the area.cpp file.....	11
Step 4: Open the Assembly File and Find the main	11
Step 5: Identify Variable Initialization	13
Step 6: Identify Arithmetic Operations in Assembly (imul)	15
Step 7: Identify Function Calls for Output (call)	15
Step 8: Wrap-Up: From C++ to CPU	16
Summary and Reflection	16
Additional Learning:.....	16
Lab 2: Exploring Control Structures in Assembly.....	17
Step 1: Write and Save the C++ Program	17
Step 2: Compile and run the area.cpp file	17
Step 3: Compile and add the -S flag to the caesercipher.cpp file.....	18
Step 4: Open the Assembly File and Find the main	18
Step 5: Identify the For Loop and the IF statements	19
Summary and Reflection	21
Additional Learning:.....	21
Lab 3a: Reversing a binary	22
Lab 3b: Reversing a binary	29

Introduction

Reverse engineering analyzes software or systems to understand their internal structure and functionality, especially when source code is unavailable. In this opening module, students are introduced to the fundamental goals and techniques of reverse engineering, which include dissecting binaries, uncovering how applications behave, and reconstructing logic from compiled code. The module also explores key real-world applications, such as malware analysis, where reverse engineering is used to identify malicious behaviors; legacy system support; software patch analysis; and improving software compatibility across systems or platforms. By grounding students in the practical value of these skills, this module sets the stage for deeper technical exploration throughout the course.

Equally important is the discussion around the legal and ethical implications of reverse engineering. Students will examine the boundaries between ethical hacking, academic research, and potential legal violations such as intellectual property infringement or violations of software licensing agreements. This module emphasizes the responsible use of reverse engineering techniques in alignment with ethical standards and applicable laws. To prepare students for the hands-on components of the course, a toolchain overview is also provided, featuring powerful tools like g++ for compilation, objdump and strings for static analysis, Ghidra and IDA Pro for disassembly and decompilation, and GDB for debugging and runtime exploration. This foundational knowledge equips learners with conceptual understanding and the practical toolkit needed to reverse engineer software responsibly and informatively.

Legal and Ethical Considerations in Reverse Engineering

Before engaging in any reverse engineering activity, it is essential to understand the legal and ethical boundaries that govern this discipline. Reverse engineering can be a powerful tool for legitimate purposes such as vulnerability discovery, malware analysis, software interoperability, and academic research. However, without proper guidance, students may inadvertently violate intellectual property laws, software licensing agreements, or digital rights management (DRM) protections. For example, reverse engineering commercial software to bypass licensing restrictions or extract proprietary algorithms may constitute a breach of copyright law or the Digital Millennium Copyright Act (DMCA) in jurisdictions like the United States. Students must also recognize that disassembling or decompiling software without the owner's consent, especially in commercial or closed-source environments, may lead to serious legal consequences.

Equally important are the ethical responsibilities that come with reverse engineering knowledge. Students must commit to using these skills in a manner that supports transparency, security, and the public good. Responsible disclosure of discovered vulnerabilities rather than exploiting them for personal gain is a core tenet of ethical hacking. Furthermore, reverse engineering should not be used to facilitate software piracy, espionage, or any activity that undermines the trust and safety of digital systems. Educational programs must emphasize the importance of integrity, consent, and compliance with all applicable laws, ensuring that students develop not just technical proficiency but also the moral framework to apply these skills ethically. This ethical grounding empowers students to contribute positively to fields like cybersecurity, forensics, and secure software development.

Understanding Computer Architecture

Understanding computer architecture is essential for practical reverse engineering, as it provides the foundation for how software interacts with hardware. Computer architecture refers to a computer system's fundamental design and organization, detailing how core components—such as the processor (CPU), memory (RAM), storage, and input/output devices—work together to execute instructions and manage data flow. This knowledge helps reverse engineers interpret how compiled binaries interact with system resources, manage memory, and perform control flow operations. By examining architectural features like instruction sets, registers, memory hierarchy, and the fetch-decode-execute cycle, students gain critical insight into how low-level code maps to hardware behavior, enabling them to analyze and deconstruct executable programs more accurately.

Computer architecture refers to a computer system's underlying design and organization, describing how hardware components such as the CPU, memory, and input/output devices interact to execute instructions and process data. In reverse engineering, understanding computer architecture is essential because it forms the basis for how compiled programs run at the lowest levels. Reverse engineers must be able to analyze how software utilizes system resources, how data is moved and stored, and how operations are executed at the hardware level. Interpreting low-level assembly code and understanding how binaries behave becomes significantly more difficult without a solid grasp of architecture.

At the heart of most modern systems lies the Von Neumann architecture, a foundational model that describes a computer with a single shared memory for both data and instructions. This architecture comprises three primary components within the CPU: the Arithmetic Logic Unit (ALU), which performs mathematical and logical operations; the Control Unit, which directs instruction flow and manages signal coordination; and Registers, which are small, fast storage locations used to hold temporary values and addresses during execution. Outside the CPU, the system relies on Random-Access

Memory (RAM) for fast-access memory and persistent storage (such as Solid-State Drives (SSDs) or Hard Disk Drives (HDDs) to save data. Input/Output (I/O) devices, such as keyboards, screens, or network interfaces, enable communication with the outside world.

Central to the functioning of the Von Neumann model is the fetch–decode–execute cycle. During this cycle, the CPU retrieves the next instruction from memory (fetch), interprets it to determine the required operation (decode), and then performs the operation (execute). This cycle continues indefinitely, enabling the execution of complex software programs. Understanding this cycle is crucial in reverse engineering because assembly instructions correspond directly to these phases, and many vulnerabilities or interesting behaviors in binaries stem from how the CPU carries out these operations. Observing how instructions move through this cycle helps reverse engineers pinpoint where to focus analysis efforts or detect anomalies.

While the Von Neumann model is widely used, it's helpful to contrast it with the Harvard architecture, which separates the memory for data and instructions. This separation allows for faster and more secure execution, commonly found in embedded systems and digital signal processors (DSPs). Unlike Von Neumann systems, where a single bus handles both data and instructions (potentially causing bottlenecks), Harvard systems use separate pathways, which can improve performance and reduce certain risks such as code injection. Understanding both models gives reverse engineers a broader context, especially when analyzing firmware or specialized systems where different architectural assumptions apply.

Memory and Registers: Stack, Heap, and CPU Internals

Understanding how memory and CPU registers operate is critical in reverse engineering, as they reveal how a program stores data, handles execution, and maintains flow control. In most modern systems, memory is divided into several segments; however, two key areas that concern reverse engineers are the **stack** and the **heap**. The **stack** is used for storing function call frames, local variables, and control data (such as return addresses). In contrast, the **heap** is used for dynamically allocated memory—data whose size or lifespan isn't known until runtime, such as objects created with `new` in C++.

The **stack** follows a Last-In, First-Out (LIFO) structure, where data is pushed onto the stack when a function is called and popped off when it returns. The operating system tightly manages it and grows downward (toward lower memory addresses). In contrast, the **heap** is a much larger area used for data that persists beyond the scope of a single function call. However, managing heap memory is more complex, as it requires the programmer to

explicitly allocate and deallocate it, which can lead to bugs such as memory leaks or use-after-free vulnerabilities.

Registers, which reside inside the CPU, are ultra-fast storage areas that hold temporary data used during processing. For instance, **RAX** (return accumulator) often stores return values, **RBP** (base pointer) and **RSP** (stack pointer) help manage stack frames, and **RCX**, **RDX**, and **RSI/RDI** may hold function arguments depending on the calling convention. During reverse engineering, examining how data flows through these registers is essential to reconstructing the function logic and understanding control flow. Disassemblers like Ghidra or objdump show how instructions interact with these registers and memory segments.



Figure 1: Memory Layout Diagram

Here's a visual table comparing key characteristics of the **stack** and **heap** memory areas:

Feature	Stack	Heap
Memory Allocation	Static	Dynamic
Size Limit	Limited	larger
Allocation Time	Fast	Slower
Deallocation	Automatic	Manual
Growth Direction	Downward	Upward
Usage Example	Function Calls	Dynamic Data

These distinctions are more than theoretical. For example, stack overflows—where too much data is pushed onto the stack—can overwrite return addresses and lead to code execution. Conversely, heap corruption can cause program crashes or enable exploitation if untrusted input manipulates memory allocation patterns. Therefore, reverse engineers must understand these regions to diagnose bugs, identify vulnerabilities, or trace malware behavior.

In summary, mastering the stack, heap, and CPU registers provides a foundation for reverse engineering. These low-level mechanics drive how software executes behind the scenes. When analyzing binaries, seeing PUSH, POP, MOV, and CALL instructions in disassembly becomes far more meaningful once the learner understands *what memory region is being accessed, which register holds what value, and how it all ties back to program logic*.

Software and Hardware Abstractions

Understanding the various levels of abstraction in software and hardware is fundamental to mastering reverse engineering. Software development begins at a high level with human-readable languages like C++ or Java, allowing developers to write complex logic in a structured and intuitive way. These high-level programs are then translated into assembly language, which is human-readable but closely aligned with the hardware's instruction set. Assembly provides a one-to-one mapping with machine code, the lowest software abstraction layer composed of binary instructions (opcodes) that the CPU can directly execute. Beneath all of this lies the microarchitecture, which includes the physical hardware components like registers, the Arithmetic Logic Unit (ALU), and execution units that carry out the instructions.

Compilers and linkers handle the transition from high-level code to executable machine code. A compiler translates high-level source code into assembly or machine code, optimizing the output for performance and compatibility with the target architecture. The linker combines object files, libraries, and other compiled resources into a single executable. This process abstracts away much of the underlying complexity from developers. Still, for reverse engineers, the challenge lies in peeling back these layers to reconstruct the program's original intent. Understanding how compilers generate specific instruction patterns or how linkers organize memory and functions helps recognize control flows and data structures when analyzing binaries.

To navigate these abstractions in reverse, practitioners use disassembly and decompilation. Disassemblers, such as Ghidra or IDA Pro, convert raw machine code into readable assembly language, revealing the program's control logic and instruction flow. Decompilers attempt to reconstruct high-level code representations, offering insights into functions, loops, and data operations. While disassembly is deterministic and precise, decompilation introduces heuristics and guesses, often resulting in incomplete or ambiguous reconstructions. Nevertheless, these tools are invaluable for reverse engineers who need to understand unknown binaries, patch vulnerabilities, or detect malicious code hidden within compiled applications.

Instruction Set Architecture (ISA) and Endianness

An Instruction Set Architecture (ISA) is a critical concept in reverse engineering, as it defines the set of instructions a processor can execute and how those instructions are encoded and processed. At its core, the ISA serves as the interface between software and hardware, specifying the machine code instructions, data types, registers, addressing modes, and memory models supported by a given CPU architecture. In reverse engineering, understanding the Instruction Set Architecture (ISA) enables analysts to accurately interpret binary instructions and relate them to the underlying hardware behaviors. Without a working knowledge of the target ISA, analyzing or reconstructing software behavior from a compiled binary is nearly impossible.

Registers are a vital component defined by the ISA and fall into several categories. General Purpose Registers (GPRs) such as RAX, RBX, RCX, and RDX are used for arithmetic, logic, and data movement operations. Special Purpose Registers include the Instruction Pointer (RIP), which tracks the next instruction to execute; the Stack Pointer (RSP), which points to the top of the current stack frame; and the Base Pointer (RBP), which helps manage function calls. The FLAGS register (RFLAGS) holds status flags used for conditional operations. In modern systems, SIMD and floating-point registers, such as XMM, YMM, and ZMM, are also used for efficient parallel operations, particularly in multimedia, scientific computation, or encryption applications.

This course focuses on the x86-64 architecture, a widely used instruction set architecture (ISA) in modern desktop and server environments. x86-64 is an extension of the older 32-bit x86 architecture, supporting a rich set of instructions and addressing modes, expanded registers, and 64-bit data types. When reverse-engineering x86-64 binaries, students must learn to recognize opcodes (the operation codes for instructions), operands (the data being acted upon), and the instruction formats used to encode them. A good grasp of these elements helps students understand how high-level operations, such as variable assignment or function calls, map to specific sequences of machine-level instructions.

Another important concept in ISA analysis is endianness, which refers to the byte order used to store multi-byte values in memory. Like most x86 processors, little-endian systems store the least significant byte first, while big-endian systems store the most significant byte first. This difference becomes crucial when interpreting memory values during reverse engineering, especially when dealing with file formats, network protocols, or mixed-architecture environments. Misinterpreting endianness can lead to incorrect conclusions about data structures and logic. A deep understanding of the ISA and its components empowers reverse engineers to confidently deconstruct and analyze compiled software, bridging the gap between binary code and its source logic.

Lab 1: Disassembling Source Code C++ to Assembly

To begin exploring how high-level C++ code maps to low-level instructions, we can start by creating a simple C++ program that calculates the area of a rectangle using the formula: $\text{Area} = \text{Length} \times \text{Width}$. The program should prompt the user for input, perform the multiplication, and then output the area to the console.

After writing the code and saving it with a .cpp extension, the program can be compiled using a compiler like g++, which converts the C++ source into machine-readable instructions.

To analyze the inner workings, we will need to compile the program with the -S flag (e.g., g++ -S area.cpp), which produces an assembly language file. This .s file reveals how the high-level logic, such as variable assignment, arithmetic operations, and output, is broken down into processor-specific instructions. Studying this file gives students a clearer understanding of how abstract code is executed at the hardware level, forming a crucial foundation for reverse engineering.

Step 1: Write and Save the C++ Program

```
1. // area.cpp
2. #include <iostream>
3.
4. using namespace std;
5.
6. int main() {
7.     int length = 5; // local variable, stack stored
8.     int width = 10; // local variable, stack stored
9.     int area = 0;
10.
11.     area = length * width;
12.     cout << "The area of the rectangle is: " << area << endl;
13.
14.     return 0;
15. }
16.
```

```

#include <iostream>

using namespace std;

int main() {
    int length=5;
    int width=10;
    int area = 0;

    area = length * width;

    cout << "The area of the rectangle is: " << area << endl;

    return 0;
}

```

Figure 2: source code of area.cpp

Step 2: Compile and run the area.cpp file

Use g++ to compile our application.

```
$g++ area.cpp -o area
```

```

mark@fedora:~/re_fun/workbook$ g++ area.cpp -o area
mark@fedora:~/re_fun/workbook$ ls -al
total 24
drwxr-xr-x. 1 mark mark    24 Feb  8 21:39 .
drwxr-xr-x. 1 mark mark   16 Feb  8 20:54 ..
-rwxr-xr-x. 1 mark mark 17224 Feb  8 21:39 area
-rw-r--r--. 1 mark mark   224 Feb  8 21:36 area.cpp
mark@fedora:~/re_fun/workbook$
mark@fedora:~/re_fun/workbook$ ./area
The area of the rectangle is: 50
mark@fedora:~/re_fun/workbook$ █

```

Figure 3: compile area.cpp

We will also want to run our application to ensure it runs correctly. We can do this by using the ./area command.

Step 3: Compile and add the -S flag to the area.cpp file

```
mark@fedora:~/re_fun/workbook$ g++ -S area.cpp
mark@fedora:~/re_fun/workbook$ ls -al
total 28
drwxr-xr-x. 1 mark mark    36 Feb  8 21:43 .
drwxr-xr-x. 1 mark mark    16 Feb  8 20:54 ..
-rwxr-xr-x. 1 mark mark 17224 Feb  8 21:39 area
-rw-r--r--. 1 mark mark   224 Feb  8 21:36 area.cpp
-rw-r--r--. 1 mark mark  1539 Feb  8 21:43 area.s
mark@fedora:~/re_fun/workbook$
```

Use g++ to compile our application.

```
$g++ -S area.cpp
```

This will compile area.cpp and generate an assembly file.

Step 4: Open the Assembly File and Find the main

_main: on macOS with clang

main: on Linux with g++

This marks the **starting point** of your program logic.

```

mark@fedora:~/re_fun/workbook$ cat area.s
.file "area.cpp"
.text
#APP
.globl _ZSt21ios_base_library_initv
.section .rodata
.align 8
.LC0:
.string "The area of the rectangle is: "
#NO_APP
.text
.globl main
.type main, @function
main:
.LFB2012:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $16, %rsp
movl $5, -4(%rbp)
movl $10, -8(%rbp)
movl $0, -12(%rbp)
movl -4(%rbp), %eax
imull -8(%rbp), %eax
movl %eax, -12(%rbp)
movl $.LC0, %esi
movl $_ZSt4cout, %edi
call _ZStlsISt11char_traitsIcEERSt13basic_ostreamIT_ES5_PKc
movq %rax, %rdx
movl -12(%rbp), %eax
movl %eax, %esi
movq %rdx, %rdi
call _ZNSolsEi
movl $_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_, %esi
movq %rax, %rdi
call _ZNSolsEPFRSoS_E
movl $0, %eax
leave
.cfi_def_cfa 7, 8

```

Step 5: Identify Variable Initialization

In the C++ code:

```
int length = 5;
int width = 10;
int area = 0;
```

In Assembly, look for **mov** instructions:

```
movl $5, -4(%rbp) ; length = 5
movl $10, -8(%rbp) ; width = 10
movl $0, -12(%rbp) ; area = 0
```

Are MOV instructions Typically Variable Assignments?

In compiled code, the mov instruction is most commonly used to represent assignments of values to variables or to transfer values between registers and memory.

Let's break down the first MOV instruction:

```
movl $5, -4(%rbp) ; length = 5
```

Breakdown:

- `movl` – This is a **"move" instruction** for a 32-bit (4-byte) value.
- `$5` – The **immediate value** (a hardcoded constant), here being the integer 5.
- `-4(%rbp)` – This is a **memory address on the stack**, 4 bytes below the base pointer `%rbp`.

In English, please: “Store the value 5 in the memory location reserved for `length`, which lives at offset -4 from the base pointer on the stack.”

Why Is This on the Stack (and Not the Heap)?

Variables like `length = 5` are local variables declared inside the `main()` function. In C++ (and C), local variables are by default stored in the stack frame of the function unless they are dynamically allocated (i.e., using `new` or `malloc`).

Variable Type	Stored In	Why?
int length = 5;	Stack	Local, automatic storage duration
int* ptr = new int(5);	Heap	Dynamically allocated
static int length = 5;	Data segment	Global/static storage

This is not on the heap because it is a non-dynamic, automatic variable.

Why Is It -4(%rbp)? What Does That Mean?

When a function starts with main(), the compiler sets up a stack frame:

%rbp (base pointer) points to the start of the frame.

Local variables are stored below the base pointer, using negative offsets like -4(%rbp) or -8(%rbp).

Function arguments (depending on the architecture and calling convention) are typically passed in registers but spilled to the stack above %rbp.

Visualizing the Stack:

HIGH ADDRESSES

```

+-----+ <-- %rbp (base pointer)
| Return Address |
| Old %rbp       |
|-----|
| length (5)     | <-- -4(%rbp)
| width (10)     | <-- -8(%rbp)
| area (0)       | <-- -12(%rbp)
+-----+

```

LOW ADDRESSES

- Stack grows **downward** (toward lower addresses).
- Local variables are given space **under** %rbp.
- The compiler assigns each variable a fixed offset (like -4) in the function prologue.

How Do We Know \$5 Is Being Moved Into That Space?

The syntax tells us: `movl $5, -4(%rbp)`

In English: Put the constant value 5 into the memory at the base pointer minus 4.

Why Use %rbp at All?

- `%rbp` is used as an **anchor point** for stack frame addressing.
- It doesn't change throughout the function (unlike `%rsp`, the stack pointer, which moves during push/pop).
- Compilers use `%rbp` so that **variables always have fixed, predictable offsets**—e.g., `length` is always `-4(%rbp)`.

This is extremely helpful when reading disassembly, as it allows you to map memory locations back to source-level variables.

SUMMARY

Concept	Value
<code>movl \$5, -4(%rbp)</code>	Store the value 5 in the stack at offset -4 from the base pointer
Stack or Heap?	Stack , because it's a local, automatic variable
Why <code>%rbp</code> ?	It's the base of the stack frame; it provides consistent offsets for variables.
Why -4?	The compiler assigned this offset to <code>length</code> in the current stack frame.

Step 6: Identify Arithmetic Operations in Assembly (`imul`)

Now let's find the multiplication logic for `area = length * width`.

Step 7: Identify Function Calls for Output (`call`)

The final part of our program prints the result. In assembly, this appears as one or more `call` instructions, corresponding to `std::cout`.

Step 8: Wrap-Up: From C++ to CPU

We've now followed the full journey of our program:

- Variables → stack (mov)
- Math → registers (imul)
- Output → function call (call)
- Return → exit (ret)

Summary and Reflection

Recapping what we have learned in Lab 1, we had the opportunity to build a simple C++ program and then have the compiler generate the assembly code for us to explore. We learned that we should try to identify the primary method since that is the main entry point into the program. We knew that typically, when we see the `movl` instruction in X86 and X86-64, we can associate it with a variable assignment.

Another key piece of assembly we have identified is the use of the 'call' keyword to display output.

C++ Code	Assembly Instruction	Meaning
<code>int length = 5;</code>	<code>movl \$5, -4(%rbp)</code>	Store value 5 in stack at offset -4
<code>area = length * width;</code>	<code>imul -8(%rbp), %eax</code>	Multiply values in memory
<code>cout << area;</code>	<code>call _ZNSolsEi</code>	Call C++ output function
<code>return 0;</code>	<code>movl \$0, %eax + ret</code>	Set return value and exit

Additional Learning:

Here is an opportunity to revisit the program and modify it to calculate the perimeter instead of the area of a rectangle. $\text{Perimeter} = 2 * \text{length} + 2 * \text{width}$;

How does the assembly change as a result of this?

Lab 2: Exploring Control Structures in Assembly

Step 1: Write and Save the C++ Program

This C++ program calculates and displays the factorial of a number entered by the user (specifically between 1 and 5).

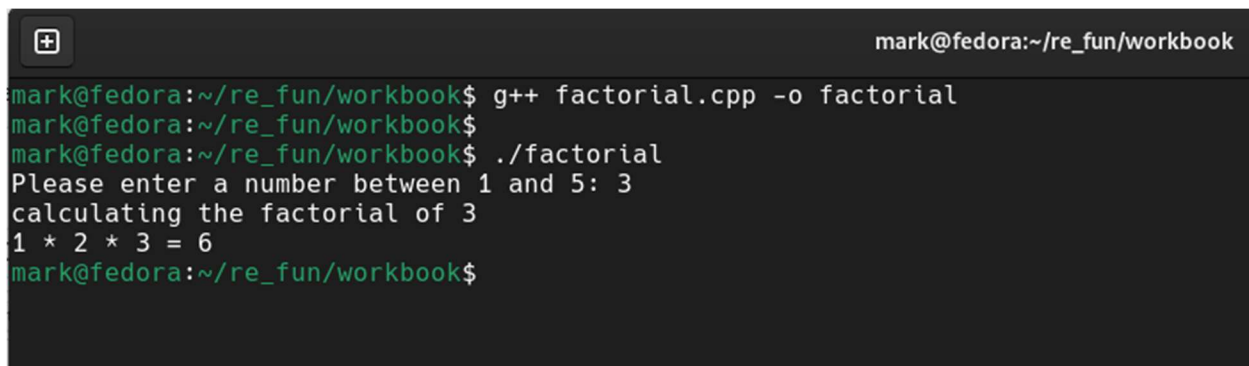
```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main() {
6.
7.     int inputNumber;
8.     int factorial=1;
9.
10.    cout << "Please enter a number between 1 and 5: ";
11.    cin >> inputNumber;
12.
13.    cout << "calculating the factorial of " << inputNumber << endl;
14.
15.    for(int i=1; i<=inputNumber; i++) {
16.
17.        factorial = factorial * i;
18.
19.        //print the output
20.        if (i != inputNumber) {
21.            cout << i << " * ";
22.        } else {
23.            cout << i << " = " << factorial << endl;
24.        }
25.    }
26. }
27.
```

Step 2: Compile and run the area.cpp file

Use g++ to compile our application.

```
$g++ factorial.cpp -o factorial
```

We will also want to run our application to ensure it runs correctly. We can do this by using the ./factorial command.

A terminal window with a dark background. The title bar shows a plus icon and the text 'mark@fedora:~/re_fun/workbook'. The terminal content shows the following commands and output:

```
mark@fedora:~/re_fun/workbook$ g++ factorial.cpp -o factorial
mark@fedora:~/re_fun/workbook$
mark@fedora:~/re_fun/workbook$ ./factorial
Please enter a number between 1 and 5: 3
calculating the factorial of 3
1 * 2 * 3 = 6
mark@fedora:~/re_fun/workbook$
```

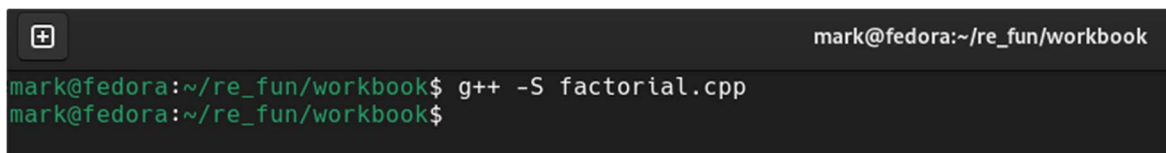
Figure 4: factorial program

Step 3: Compile and add the -S flag to the caesercipher.cpp file

Use g++ to compile our application.

```
$g++ -S factorial.cpp
```

This will compile factorial.cpp and generate an assembly file.

A terminal window with a dark background. The title bar shows a plus icon and the text 'mark@fedora:~/re_fun/workbook'. The terminal content shows the following commands:

```
mark@fedora:~/re_fun/workbook$ g++ -S factorial.cpp
mark@fedora:~/re_fun/workbook$
```

Figure 5 A factorial program generates assembly

Step 4: Open the Assembly File and Find the main

_main: on macOS with clang

main: on Linux with g++

This marks the **starting point** of your program logic.

Step 5: Identify the For Loop and the IF statements

Being able to identify loops and conditional statements is key to understanding how an application works. In assembly language, loops are typically implemented using labels and conditional or unconditional jump instructions. Here's how you can spot them:

1. **Labels:** These are markers in the code that serve as jump destinations. They usually start with `.L` followed by a number (e.g., `.L2`, `.L3`).
2. **Jump Instructions:** Instructions like `jmp`, `je` (jump if equal), `jne` (jump if not equal), `jg` (jump if greater), `jl` (jump if less), etc., are used to control the flow of the program.
3. **Comparison Instructions:** The `cmp` instruction is used to compare two values and set the flags accordingly for conditional jumps.

In your `factorial.cpp`, the loop is implemented as:

```
1. for(int i = 1; i <= inputNumber; i++) {  
2.     factorial = factorial * i;  
3.     // Output code  
4. }  
5.
```

In the corresponding assembly (`factorial.s`), let's look at the following code segment:

```
47. movl    $1, -8(%rbp)      ; i = 1  
48. jmp     .L2              ; jump to loop condition  
49. .L5:                ; loop body starts here  
50. movl    -4(%rbp), %eax    ; load factorial  
51. imull   -8(%rbp), %eax    ; multiply by i  
52. movl    %eax, -4(%rbp)    ; store result back into factorial  
....  
81. addl    $1, -8(%rbp)      ; i++  
82. .L2:                ; loop condition check  
83. movl    -12(%rbp), %eax   ; load inputNumber  
84. cmpl    %eax, -8(%rbp)    ; compare i to inputNumber  
85. jle     .L5              ; if i <= inputNumber, jump back to .L5
```

This is the for loop!

`.L2` is the label where we check if `i <= inputNumber`.

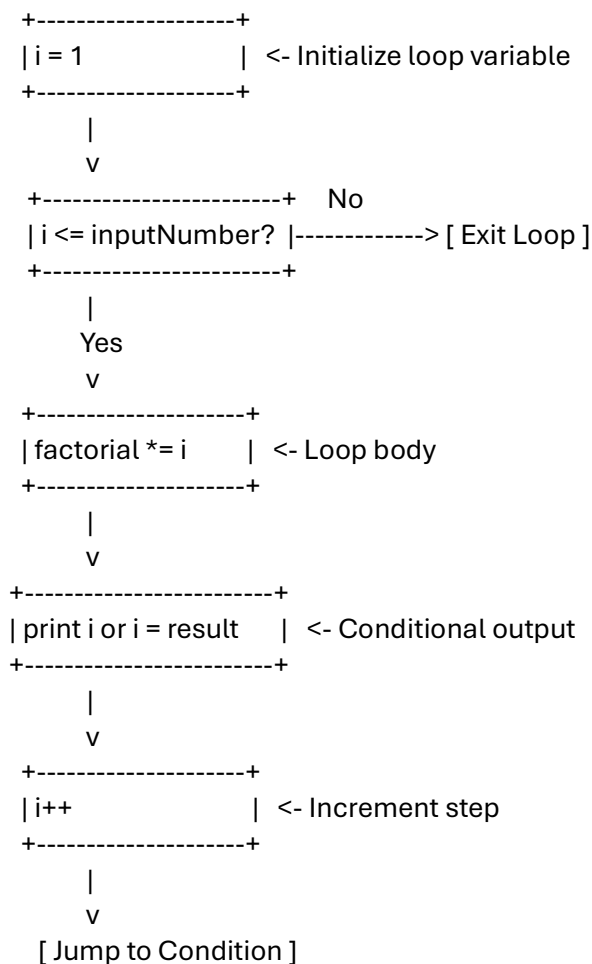
`.L5` is the loop body — the part that multiplies and prints.

`addl $1, -8(%rbp)` increments `i`.

The `jle .L5` jumps back if `i` is still within bounds, keeping the loop going.

When you write a loop in C++, the compiler turns it into a series of jumps and comparisons in assembly. Think of it like a choose-your-own-adventure book: at each label, the program decides what to do next, such as repeat, skip, or exit.

An ASCII visual graph helps beginners understand how a program's logic flows at the assembly level by turning abstract jumps and labels into a clear, step-by-step diagram. It makes it easier to visualize how loops, conditionals, and function calls connect, especially when labels like .L2 and jmp might otherwise seem confusing. For reverse engineering, these diagrams bridge the gap between high-level C++ logic and low-level machine behavior, helping students mentally map control flow even when the source code is unavailable.



Understanding these jump patterns is critical in reverse engineering. You might not see a for loop in binary, but you will see jump instructions like jle, jmp, cmp, and labels like .L2. Once you learn to read them like road signs, you can mentally reconstruct the logic of the original program—even if you don't have the source code.

Summary and Reflection

- Look for `movl`, `cmp`, `jmp`, `je`, or `jle` to find loop logic.
- Recognize variable offsets like `-4(%rbp)` for factorial, `-8(%rbp)` for `i`, and `-12(%rbp)` for `inputNumber`.
- Understand labels like `.L2`, `.L3`, `.L5` as "goto" markers the assembly jumps to.
- This is how a simple for loop and variable manipulations look when translated into low-level x86_64 assembly.

Additional Learning:

- Can you identify where the loop variable `i` is incremented in the loop?
- What label marks the loop body?

Lab 3a: Reversing a binary

Now that we have a fundamental understanding of assembly, we are ready to download a binary and attempt to reverse-engineer it. For this lab, we will download the binary from the GitHub repository for Lab3a and use GDB to explore it.

Let's download the lab3a binary from my GitHub and start to explore our binary:

```
mark@fedora:~/re_fun/workbook$ ls -al lab3a
-rwxr-xr-x. 1 mark mark 17224 Jun  8 21:33 lab3a
mark@fedora:~/re_fun/workbook$
```

Let's fire up GDB and see what we can figure out about this binary.

Syntax: `$gdb <filename>`

```
mark@fedora:~/re_fun/workbook$ ls -al lab3a
-rwxr-xr-x. 1 mark mark 17224 Jun  8 21:33 lab3a
mark@fedora:~/re_fun/workbook$ gdb lab3a
GNU gdb (Fedora Linux) 16.2-1.fc40
Copyright (C) 2024 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 "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://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 lab3a...

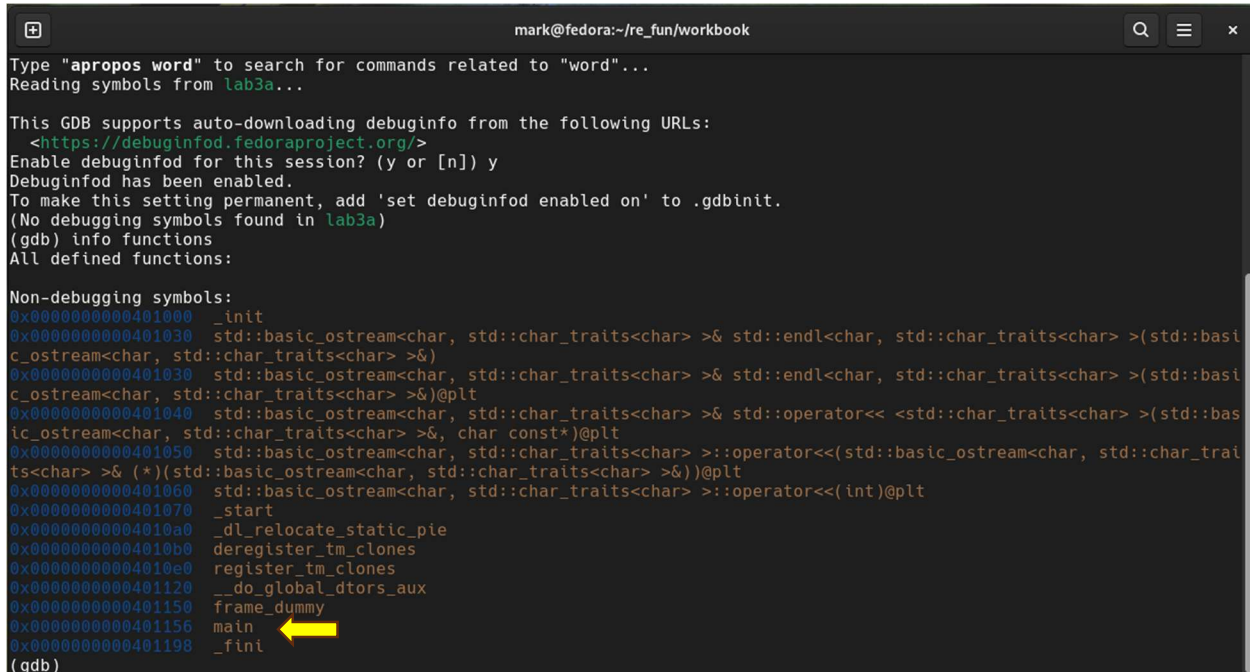
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
(No debugging symbols found in lab3a)
(gdb)
```

Figure 6 start GDB for lab3a binary

Let's start by seeing what functions are in this binary.

Syntax: (gdb) info functions

This will provide a list of the functions in the file.

A terminal window titled 'mark@fedora:~/re_fun/workbook' showing the output of the GDB command 'info functions'. The output lists non-debugging symbols including _init, various std::basic_ostream and std::operator functions, _start, _dl_relocate_static_pie, deregister_tm_clones, register_tm_clones, __do_global_dtors_aux, frame_dummy, main (highlighted with a yellow arrow), and _fini.

```
mark@fedora:~/re_fun/workbook
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab3a...

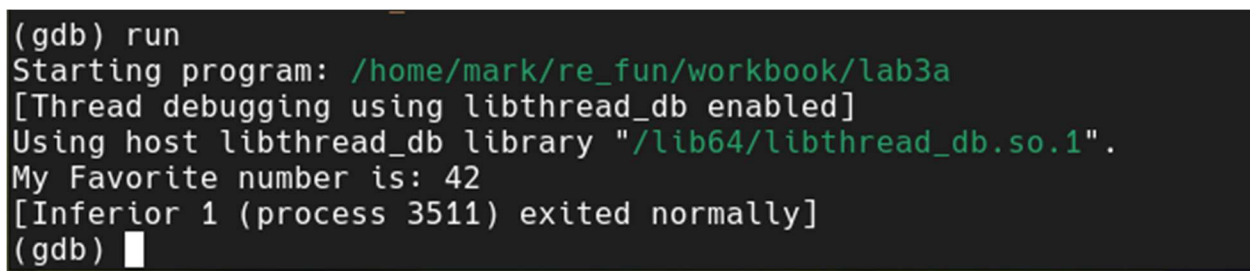
This GDB supports auto-downloading debuginfo from the following URLs:
<https://debuginfod.fedoraproject.org/>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
(No debugging symbols found in lab3a)
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000401000 _init
0x0000000000401030 std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::bas
c_ostream<char, std::char_traits<char> >&)
0x0000000000401030 std::basic_ostream<char, std::char_traits<char> >& std::endl<char, std::char_traits<char> >(std::bas
c_ostream<char, std::char_traits<char> >&)@plt
0x0000000000401040 std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::char_traits<char> >(std::bas
ic_ostream<char, std::char_traits<char> >&, char const*)@plt
0x0000000000401050 std::basic_ostream<char, std::char_traits<char> >::operator<<(std::basic_ostream<char, std::char_trai
ts<char> >& (*) (std::basic_ostream<char, std::char_traits<char> >&))@plt
0x0000000000401060 std::basic_ostream<char, std::char_traits<char> >::operator<<(int)@plt
0x0000000000401070 _start
0x00000000004010a0 _dl_relocate_static_pie
0x00000000004010b0 deregister_tm_clones
0x00000000004010e0 register_tm_clones
0x0000000000401120 __do_global_dtors_aux
0x0000000000401150 frame_dummy
0x0000000000401156 main
0x0000000000401198 _fini
(gdb)
```

Figure 7 (gdb) info functions

We can see that this file only has a main function.

Let's run it and see what happens.

A terminal window showing the output of the GDB command 'run'. The program starts, prints 'My Favorite number is: 42', and exits normally.

```
(gdb) run
Starting program: /home/mark/re_fun/workbook/lab3a
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
My Favorite number is: 42
[Inferior 1 (process 3511) exited normally]
(gdb)
```

Figure 8 (gdb) run

Interesting. It provides some output.

We know there is a main() function, so let's disassemble the main() function.

Syntax: (gdb) disassemble <function>


```

(gdb) disassemble main
Dump of assembler code for function main:
0x0000000000401156 <+0>:    push    %rbp
0x0000000000401157 <+1>:    mov     %rsp,%rbp
0x000000000040115a <+4>:    sub     $0x10,%rsp
0x000000000040115e <+8>:    movl    $0x2a,-0x4(%rbp)
0x0000000000401165 <+15>:   mov     $0x402010,%esi
0x000000000040116a <+20>:   mov     $0x404040,%edi
0x000000000040116f <+25>:   call    0x401040 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
0x0000000000401174 <+30>:   mov     %rax,%rdx
0x0000000000401177 <+33>:   mov     -0x4(%rbp),%eax
0x000000000040117a <+36>:   mov     %eax,%esi
0x000000000040117c <+38>:   mov     %rdx,%rdi
0x000000000040117f <+41>:   call    0x401060 <_ZNSolsEi@plt>
0x0000000000401184 <+46>:   mov     $0x401030,%esi
0x0000000000401189 <+51>:   mov     %rax,%rdi
0x000000000040118c <+54>:   call    0x401050 <_ZNSolsEPFRSoS_E@plt>
0x0000000000401191 <+59>:   mov     $0x0,%eax
0x0000000000401196 <+64>:   leave
0x0000000000401197 <+65>:   ret
End of assembler dump.
(gdb)

```

Figure 9 (gdb) disassemble main() function

Step 1: Understand Function Prologue

```

0x0000000000401156 <+0>:    push    %rbp
0x0000000000401157 <+1>:    mov     %rsp,%rbp

```

- **What it does:** Sets up the stack frame.
- %rbp (base pointer) is saved and then re-pointed to the current stack pointer (%rsp).
- This is **standard boilerplate** for function prologues in x86_64 calling conventions.

Step 2: Allocate Space for Local Variables

```

0x000000000040115a <+4>:    sub     $0x10,%rsp

```

What it does: Reserves 16 bytes on the stack.

NOTE: The hexadecimal number 0x10 is equivalent to 16 in decimal.

Step 3: Identify Local Variables

```

0x000000000040115e <+8>:    movl    $0x2a,-0x4(%rbp)

```

What's happening: The immediate value 0x2a (42 in decimal) is stored at -0x4(%rbp).

This is the first and only clearly visible local variable. We only see one *movl* opcode.

Remember that *movl* is a sign of variable assignment on the stack.

Step 4: Output the Local Variable

```
0x401174: mov  %rax,%rdx
0x401177: mov  -0x4(%rbp),%eax
0x40117a: mov  %eax,%esi
0x40117c: mov  %rdx,%rdi
0x40117f: call 0x401060 ...@plt
```

- Loads the local variable x (from -0x4(%rbp)) into %eax, then moves it to %esi.
- %rdx holds the output stream and is moved into %rdi.
- Calls a function like std::cout << x;.

Step 5: Final Output and Return

```
0x401184: mov  $0x401030,%esi
0x401189: mov  %rax,%rdi
0x40118c: call 0x401050 <...@plt>
```

Likely outputs std::endl or flushes the stream.

```
0x401191: mov  $0x0,%eax
0x401196: leave
0x401197: ret
```

- leave restores the previous stack frame.
- ret exits the function, returning 0 from main().

That was a lot of fun exploring the binary with GDB! Using live debugging to peek into memory, follow register values, and disassemble functions step by step provides a powerful, hands-on way to understand how software works under the hood.

Let's exit GDB by typing: (gdb) quit

Now that we've seen how GDB works interactively, let's try another way to disassemble our binary statically using a tool called objdump.

What Is objdump?

objdump is a command-line utility that lets us disassemble binaries and inspect their contents without running them. It's excellent for creating a full snapshot of a program's assembly, and it's especially useful when you want to analyze code offline or share your findings with others. Let's use objdump to disassemble the lab3a binary and save the output to a file:

```
mark@fedora:~/re_fun/workbook$ objdump -d -M intel lab3a > disasm.txt
mark@fedora:~/re_fun/workbook$
```

Flag Breakdown:

- -d: Disassemble the binary's code section (usually .text).
- -M intel: Display the assembly using Intel syntax (which is more readable for many people than AT&T syntax).
- lab3a: The binary file to be disassembled.
- > disasm.txt: Redirects the output into a text file named disasm.txt so you can view or search it easily later.

```
mark@fedora:~/re_fun/workbook$ cat disasm.txt
lab3a:      file format elf64-x86-64

Disassembly of section .init:

0000000000401000 <_init>:
401000:  f3 0f 1e fa      endbr64
401004:  48 83 ec 08      sub    rsp,0x8
401008:  48 8b 05 d1 2f 00 00 mov    rax,QWORD PTR [rip+0x2fd1]      # 403fe0 <__gmon_start__@Base>
40100f:  48 85 c0          test   rax,rax
401012:  74 02            je     401016 <_init+0x16>
401014:  ff d0            call   rax
401016:  48 83 c4 08      add    rsp,0x8
40101a:  c3              ret

Disassembly of section .plt:

0000000000401020 <_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_@plt-0x10>:
401020:  ff 35 ca 2f 00 00 push   QWORD PTR [rip+0x2fca]      # 403ff0 <_GLOBAL_OFFSET_TABLE_+0x8>
401026:  ff 25 cc 2f 00 00 jmp     QWORD PTR [rip+0x2fcc]      # 403ff8 <_GLOBAL_OFFSET_TABLE_+0x10>
40102c:  0f 1f 40 00      nop    DWORD PTR [rax+0x0]

0000000000401030 <_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_@plt>:
401030:  ff 25 ca 2f 00 00 jmp     QWORD PTR [rip+0x2fca]      # 404000 <_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_@GLIBCXX_3.4>
401036:  68 00 00 00 00 00 push   0x0
40103b:  e9 e0 ff ff ff   jmp     401020 <_init+0x20>

0000000000401040 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>:
401040:  ff 25 c2 2f 00 00 jmp     QWORD PTR [rip+0x2fc2]      # 404008 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@GLIBCXX_3.4>
401046:  68 01 00 00 00 00 push   0x1
40104b:  e9 d0 ff ff ff   jmp     401020 <_init+0x20>

0000000000401050 <_ZNSolsEPFRSoS_E@plt>:
401050:  ff 25 ba 2f 00 00 jmp     QWORD PTR [rip+0x2fba]      # 404010 <_ZNSolsEPFRSoS_E@GLIBCXX_3.4>
401056:  68 02 00 00 00 00 push   0x2
40105b:  e9 c0 ff ff ff   jmp     401020 <_init+0x20>
```

```
mark@fedora:~/re_fun/workbook

401136: c6 05 13 30 00 00 01 mov BYTE PTR [rip+0x3013],0x1 # 404150 <completed.0>
40113d: 5d pop rbp
40113e: c3 ret
40113f: 90 nop
401140: c3 ret
401141: 66 66 2e 0f 1f 84 00 data16 cs nop WORD PTR [rax+rax*1+0x0]
401148: 00 00 00 00
40114c: 0f 1f 40 00 nop DWORD PTR [rax+0x0]

0000000000401150 <frame_dummy>:
401150: f3 0f 1e fa endbr64
401154: eb 8a jmp 4010e0 <register_tm_clones>

0000000000401156 <main>:
401156: 55 push rbp
401157: 48 89 e5 mov rbp,rbp
40115a: 48 83 ec 10 sub rsp,0x10
40115e: c7 45 fc 2a 00 00 00 mov DWORD PTR [rbp-0x4],0x2a
401165: be 10 20 40 00 mov esi,0x402010
40116a: bf 40 40 40 00 mov edi,0x404040
40116f: e8 cc fe ff ff call 401040 <_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc@plt>
401174: 48 89 c2 mov rdx,rax
401177: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]
40117a: 89 c6 mov esi,eax
40117c: 48 89 d7 mov rdi,rdx
40117f: e8 dc fe ff ff call 401060 <_ZNSolsEi@plt>
401184: be 30 10 40 00 mov esi,0x401030
401189: 48 89 c7 mov rdi,rax
40118c: e8 bf fe ff ff call 401050 <_ZNSolsEPFRSoS_E@plt>
401191: b8 00 00 00 00 mov eax,0x0
401196: c9 leave
401197: c3 ret

Disassembly of section .fini:

0000000000401198 <_fini>:
401198: f3 0f 1e fa endbr64
40119c: 48 83 ec 08 sub rsp,0x8
4011a0: 48 83 c4 08 add rsp,0x8
4011a4: c3 ret

mark@fedora:~/re_fun/workbook$
```

Great, we now have two ways to generate the assembly from our binary. Let's explore a few other tools.

What Does the strings Command Do?

The strings command scans a binary (or any file) and extracts printable ASCII (or Unicode) strings embedded within it. This is especially useful in reverse engineering because it helps you:

- Find hardcoded messages, such as error prompts, usernames, or debug output.
- Discover file paths, URLs, or commands that might be executed.
- Get a quick peek at what the binary might be doing, without disassembling a single instruction.

Syntax: `$ strings lab3a`

```
mark@fedora:~/re_fun/workbook$ strings lab3a
/lib64/ld-linux-x86-64.so.2
;PV|%b(cs)
__gmon_start__
_ZNSolsEi
_ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_0_ES6_
_ZNSolsEPFRSoS_E
_ZSt21ios_base_library_initv
_ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
_ZSt4cout
__libc_start_main
libstdc++.so.6
libm.so.6
libgcc_s.so.1
libc.so.6
GLIBC_2.34
GLIBCXX_3.4.32
GLIBCXX_3.4
PTE1
H=@@
My Favorite number is:
;*3$"
GCC: (GNU) 14.2.1 20240912 (Red Hat 14.2.1-3)
AV:4p1260
RV:running gcc 14.2.1 20240912
BV:annobin gcc 14.2.1 20240912
GW:0x7d60562 ../sysdeps/x86/abi-note.c
SP:3
SC:1
CF:8 ../sysdeps/x86/abi-note.c
FL:0 ../sysdeps/x86/abi-note.c
GA:1
PI:4
SE:0
iS:0
GW:0x7d60562 init.c
CF:8 init.c
FL:0 init.c
GW:0x7d60562 static-reloc.c
SP:0 static-reloc.c
CF:8 static-reloc.c
```

Take note of any phrases that look like output text, filenames, or function names — we'll try to match those with instructions when we analyze the disassembly next!

Lab 3b: Reversing a binary

Now that we have a fundamental understanding of assembly, we are ready to download a binary and attempt to reverse-engineer it. For this lab, we will download the binary from the GitHub repository for Lab3b and use GDB to explore it.