CHAPTER 1

Decompilation and Architecture

An effective reverse engineer or cracker is one who understands the systems they are analyzing. Software is designed to run in a particular environment, and if you don't understand how that environment works, you will struggle to understand the software.

This chapter explores the steps necessary to get started reverse engineering an application. Decompilation is crucial to transforming an application from machine code to something that can be read and understood by humans. To actually analyze the resulting code, it is also necessary to understand the architecture of the computers that it is designed to run on.

Decompilation

Most programmers write using a higher-level programming language like C/C++ or Java, which is designed to be human-readable. However, computers are designed to run machine code, which represents instructions in binary.

Compilation is the process of converting a programming language to machine code. This means *decompilation* would be the process of taking machine code back to the original programming language, recovering the original source code. When available, this is the easiest approach to reverse engineering because source code is designed to be read and interpreted by a human. The majority of this

book will focus on the more typical case when decompilation is not possible. But for the purposes of learning, it is important to understand that sometimes you can decompile back to the source code, and when that is an option, you should take it.

When Is Decompilation Useful?

For many programming languages, full decompilation is impossible. These languages build code directly to machine code, and some information, such as variable names, is lost in the process. While some advanced decompilers can build pseudocode for these languages, the process isn't perfect.

However, some programming languages use what's called *just-in-time* (*JIT) compilation*. When programs written in JIT languages are "built," they are converted from the source code into an *intermediate language (IL)*, not machine code. JIT compilers store a copy of the code in this IL until the program is run, at which point the code is converted to machine code. Examples of JIT languages include Java, Dalvik (Android), and .NET.

For example, Java is well-known for being largely platform-agnostic, and the reason for this is its use of an IL (Java bytecode) and the Java Virtual Machine (JVM). By distributing the program code as bytecode and compiling it only at runtime, Java's JVM translates from the Java IL to machine code specific to the machine it's running on. While this approach can negatively impact file size and performance, it pays off in portability.

JIT compilation also makes reverse engineering these applications much easier. These intermediate languages are similar enough to the original source code that they can be decompiled or converted back into usable source code. Source code is designed to be human-readable, making it far easier to understand the application's logic and identify software protections or other embedded secrets.

Decompiling JIT Programming Languages

For JIT languages like .NET, several free decompilers are available. One widely used .NET decompiler is JetBrains dotPeek, which is available from `www.jetbrains.com/decompiler`. Figure 1.1 shows an example of .NET code decompiled in dotPeek.

As shown in the figure, the .NET code is easily readable after decompilation because the intermediate language encodes a wealth of information as metadata, enabling more accurate reconstruction of the source code. Any sensitive information or trade secrets contained within the code are easily accessible to a reverse engineer.

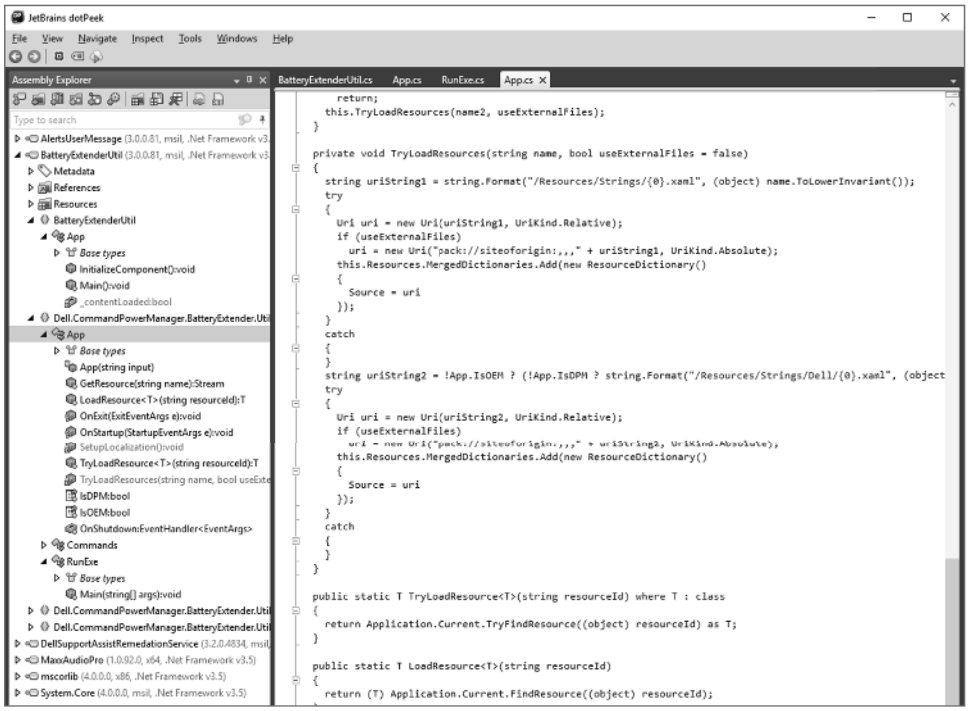

Figure 1.1: JetBrains dotPeek .NET decompiler

Defending JIT Languages

Unlike true machine code programs, JIT-compiled programs can often be converted to source code. Lowering the bar for reverse engineering the code makes many of the x86 anti-reverse engineering defenses discussed in later chapters unnecessary and overkill.

For decompilable languages, a commonly used defense against reverse engineering is obfuscation. Figure 1.2 shows an example of a .NET application before and after obfuscation.

The top half of the figure contains code before obfuscation occurs, where the function and variable names and strings are easily readable. The information in these variable names makes it easier for a reverse engineer to understand the purpose of each function and how the application works as a whole.

In the bottom half of the image, we see the obfuscated version of the same code. Now, function names, variable names, and strings are all mangled, making it much harder to understand the purpose of the function shown, let alone the application as a whole.

Another important security best practice is to avoid writing security or privacy-critical code in JIT languages where reverse engineering is easy. Instead, write

Figure 1.2: Obfuscation in JetBrains dotPeek

this code in an assembled language, such as C/C++, where reverse engineering is significantly more difficult. This code can be included in DLLs that are linked to the executable containing the nonsensitive code written in a JIT language.

Lab 1: Decompiling

This is the first hands-on lab for this book. Labs and all associated instructions can be found in their corresponding folder here:

`https://github.com/DazzleCatDuo/X86-SOFTWARE-REVERSE-ENGINEERING-CRACKING-AND-COUNTER-MEASURES`

For this lab, please locate Lab Decompiling and follow the provided instructions.

Skills to Practice

Every lab in this book is designed to teach and provide hands-on experience with certain skills. This lab's skills to practice include the following:

- Decompiling
- Performing introductory reverse engineering

To learn these skills, you'll be using JetBrains dotPeek to reverse engineer and modify a .NET application.

Takeaways

Decompiling is a powerful and easy approach to understanding and modifying a program. However, it doesn't work on every program. While programs written in languages such as C/C++ can be decompiled using tools such as IDA's Hex-Rays Decompiler or Ghidra, the result is often low-quality and difficult to use.

When developing applications that contain sensitive information or that you don't want modified, it's better to use a language that isn't easily decompiled. For example, C/C++ is a better choice for sensitive functionality than a .NET language such as C#.

Architecture

Decompilation is the easy approach to reverse engineering because it gets you back to higher-level languages and logic structures. However, this easy path is not often available. For languages that build to machine code, we need to go deeper and understand how computer architectures and machine and assembly code work.

Computer Architecture

It's generally thought that the average programmer doesn't need an in-depth understanding of how computers work. When writing a program in a procedural language, the operating system handles all of the low-level operations. A program is displayed as a process that has access to the processor, memory, and file system whenever it needs them. Processes appear to have their own contiguous memory spaces, and files are just a sequence of bytes to read and write.

However, none of this is actually true, and your operating system has been abstracting the truth from you (to make it easier to program). A solid understanding of how computer architecture actually works is essential for a reverse engineer. Figure 1.3 shows the main components that make up a computer, including the central processing unit, bridge, memory, and peripherals.

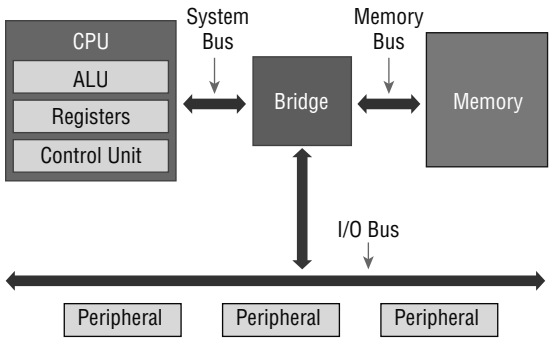

Figure 1.3: Computer architecture

The Central Processing Unit

The *central processing unit (CPU)* is where processing occurs on a computer. Inside the CPU are the following components:

- **Arithmetic logic unit (ALU):** The ALU performs mathematical operations within the computer, such as addition and multiplication.
- **Registers:** Registers perform temporary data storage and are used as the primary inputs and outputs of x86 instructions. Registers provide extremely fast access to a single word of data and are typically accessed by name.
- **Control units:** Control units execute code. This includes reading instructions and orchestrating the operations of other elements within a computer.

Bridges and Peripherals

The CPU is connected via a *bus* to a *bridge*. The purpose of the bridge is to connect the CPU to other components of the system, including memory and the *I/O bus*, which is where peripherals such as the keyboard, mouse, and speakers are connected to the system. While information flows over a bus, the bridge is responsible for controlling this traffic and ensuring that traffic flowing in over one bus is routed out over the appropriate bus.

Peripherals, connected via the I/O bus, allow the computer to communicate with the outside world. This includes sending and receiving data from the graphics card, keyboard, mouse, speakers, and other systems.

Memory and Registers

As its name suggests, *memory* is where data is stored on the computer. Data is stored as a linear series of bytes that are accessed via their address. This design allows moderately fast access to data stored on the system.

When a program wants to access data in memory, the CPU sends a request via a bus to the bridge, which forwards it to the memory, where the data at the indicated address is accessed. The requested data then needs to retrace that route and return to the CPU before it can be used by the program. In contrast, a register is physically located within the CPU, making it far more accessible.

Registers are storage that lives inside of the CPU and, unlike memory, are not a linear series of bytes. Registers are specifically named and have set sizes associated with each.

Registers and memory both serve the same purpose: they store data. However, they have different specializations (quality versus quantity). Registers are few in number and expensive, but they provide extremely fast access to data. Memory is cheap and plentiful but offers slower access speeds.

The bulk of the data associated with a program, the code itself and its data, will be stored in memory. While the program is running, small chunks of data will be copied to the registers for processing.

Assembly

Computers run on binary, digital logic. Everything is either on (1) or off (0). This includes programs running on a computer. All high-level languages are eventually converted into a series of bits called *machine code*. This machine code defines the set of instructions that the computer executes to perform a desired function.

Introduction to Machine Code

Every programmer begins learning a language with a "hello world" program. In x86, the machine code for "hello world" is as follows:

```
55 89 e5 83 e4 f0 83 ec 10 b8 b0 84 04 08 89 04 24 e8 1a ff ff ff b8 00
00 00 00 c9 c3 90
```

This machine code is written in hexadecimal for readability and compactness, but its true value is a binary string of 1s and 0s. This binary string contains instructions to flip transistors to calculate information, fetch data from memory, send signals over the system buses, interact with the graphics card, and, finally, print out the "hello world" text. If this string of characters seems a bit short to accomplish all this, it's because these instructions trigger the operating system (in this example Linux) to help out.

Machine code controls the processor at the most detailed possible level. Some of the functions that machine code performs include the following:

- Moving data in and out of memory
- Moving data to and out of registers
- Controlling the system bus
- Controlling the ALU, control unit, and other components

This low-level control means that applications written in machine code can be incredibly powerful and efficient. However, while memorizing and inputting various series of bits to perform certain tasks is pretty awesome, it is inefficient and prone to error.

From Machine Code to Assembly

In machine code, a series of bits represents a particular action. For example, `0x81` or `10000001` is an instruction that adds two values together and stores the result at a particular location.

Assembly code is designed to be a human-readable version of machine code. Instead of memorizing a binary or hexadecimal string like `0x81` or `10000001`, a programmer can use `add`. The `add` mnemonic is mapped to `0x81`, so this shorthand makes programming easier without losing any of the benefits of writing in machine code.

Translating machine code to assembly code makes it much easier to understand. For example, the previous "hello world" example code can be converted into a series of comprehensible instructions.

MACHINE CODE	ASSEMBLY
`55`	`push ebp`
`89 e5`	`mov ebp,esp`
`83 e4 f0`	`and esp, 0xfffffff0`
`83 ec 10`	`sub esp, 0x10`
`b8 b0 84 04 08`	`mov eax`
`89 04 24`	`mov [esp], eax`
`e8 1a ff ff ff`	`call 80482f4`
`b8 00 00 00 00`	`mov eax, 0x0`
`c9`	`leave`
`c3`	`ret`
`90`	`nop`

If you understand machine code, writing directly in it can be fun, and there are cases where it may make sense. However, the majority of the time, it is inefficient and impractical. Writing in assembly provides the same benefits as writing in machine code but is much more practical.

After code has been written in assembly, it can be translated to machine code by an *assembler* in a process called *assembling*. A program already in machine code can be disassembled into assembly code by a disassembler.

DEFINITION

Assemblers convert assembly code to machine code. Disassemblers convert machine code to assembly.

Many programmers don't write in machine code or assembly. Instead, they use higher-level languages that abstract away more of the details. For example, the following pseudocode is similar to many high-level procedural languages.

```
int x=1, y=2, z=x+y;
```

During the compiling process, these higher-level languages are converted into assembly code similar to the following:

```
mov   [ebp-4], 0x1
mov   [ebp-8], 0x2
mov   eax, [ebp-8]
mov   edx, [ebp-4]
lea   eax, [edx+1*eax]
mov   [ebp-0xc], eax
```

An assembler can then be used to convert the assembly code into the following machine code that a computer can use:

```
c7 45 fc 01 00 00 00 c7 45 f8 02 00 00 00 8b 45 f8 8b 55 fc 8d 04 02
89 45 f4
```

Instruction Set Architectures and Microarchitectures

The word *computer* covers a wide range of systems. A smartwatch and a desktop computer both work in similar ways. However, their internal components can differ significantly.

An *instruction set architecture (ISA)* describes the ecosystems where programs run. Some of the factors that an ISA defines include the following:

- **Registers:** The ISA specifies whether a processor has a single register or hundreds. It also defines the size of these registers, whether they contain 8 bits or 128 bits.
- **Addresses and data formats:** The ISA specifies the format for addresses used to access data in memory. It also defines how many bytes the system can grab from memory at a time.
- **Machine instructions:** Different ISAs may support different sets of instructions. The ISA defines whether addition, subtraction, equality, halt, and other instructions are supported.

By defining the capabilities of the physical system, the ISA also indirectly defines the assembly language. The ISA specifies which low-level instructions are available and what those instructions do.

A microarchitecture describes how a particular ISA is implemented on a processor. Figure 1.4 shows an example of the Intel Core 2 architecture.

Together, an ISA and microarchitecture define the *computer architecture*. The existence of thousands of ISA and thousands of microarchitectures means that there are thousands of computer architectures as well.

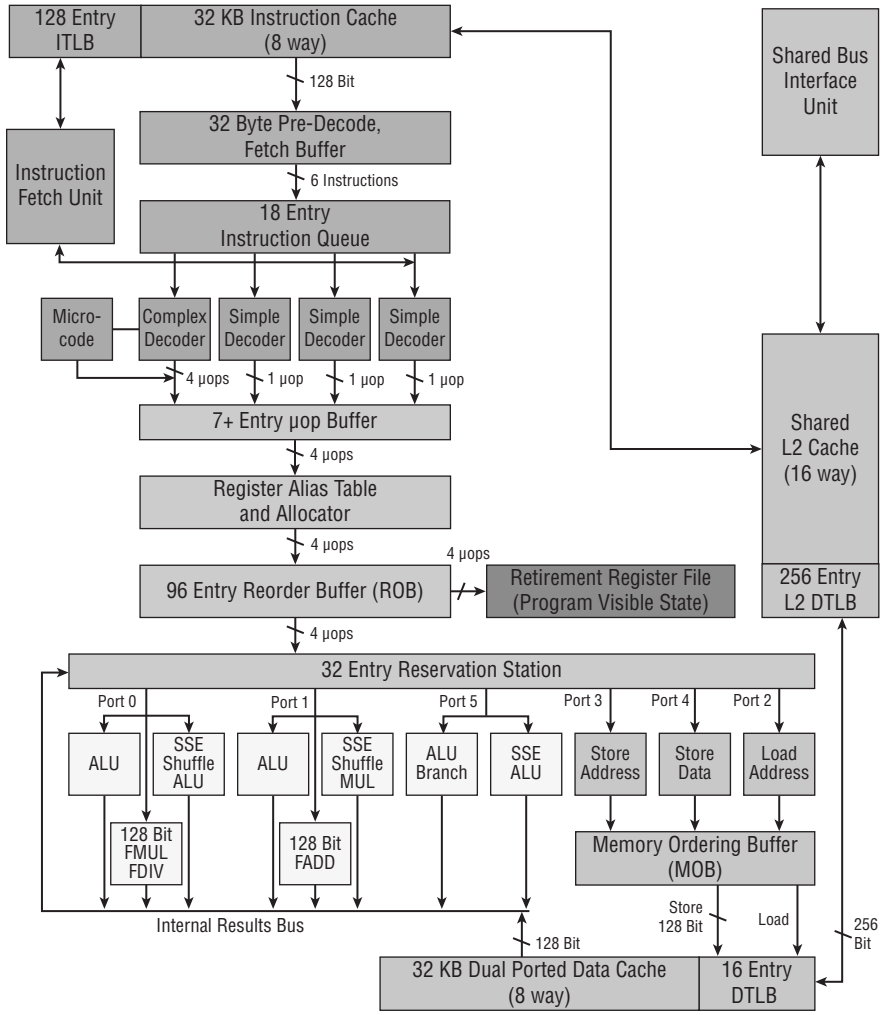

Figure 1.4: Intel Core 2 architecture

DEFINITION

An instruction set architecture defines how registers, addresses, data formats, and machine instructions work. Microarchitectures implement ISAs on a processor. Together, an ISA and microarchitecture define a computer architecture.

RISC vs. CISC Computer Architectures

While thousands of computer architectures exist, they can be broadly divided into two main categories. Reduced instruction set computing (RISC) architectures

define a small number of simpler instructions. In general, RISC architectures are cheaper and easier to create, and the hardware is physically smaller and consumes less power.

In contrast, a complex instruction set computing (CISC) architecture defines a larger number of more powerful instructions. CISC processors are more expensive and difficult to create and are typically larger and consume more power.

While CISC architectures may seem objectively worse than RISC ones, their main benefit lies in the ease and efficiency of programming. For example, consider a hypothetical example where a program wants to multiply a value by 5 in a RISC versus CISC system.

CISC	RISC
`mul [100], 5`	`load r0, 100`
	`mov r1, r0`
	`add r1, r0`
	`add r1, r0`
	`add r1, r0`
	`add r1, r0`
	`mov [100], r1`

In this example, a CISC processor can perform the calculation in a single instruction if it has a multiplication operation that can load a value from memory, multiply it, and store the result at the same memory location. However, a RISC processor may lack a multiplication operator because it is a complex operation. Instead, the RISC loads the value from memory, adds it to itself four times, and stores the result in the same memory location across seven steps.

RISC and CISC architectures both have their advantages, disadvantages, and use cases. For example, a RISC operator may take 100 instructions to perform the same operation that a CISC operator can perform in one. However, that single CISC operation may take 100× as long to run or 100× the power.

Both RISC and CISC instruction sets are in common use today. Some examples of widely used RISC architectures include the following:

- ARM (used by phones, tablets)
- MIPS (used by embedded systems and networking equipment)
- PowerPC (used by original Macs and Xbox360)

In this book, we focus on the x86 assembly language, which is a CISC architecture. This architecture is in use on all modern PCs and servers and is supported by all the main operating systems (Windows, Mac, Linux) and even

some gaming systems, such as the Xbox One. Making it one of the most powerful to learn for software cracking.

Summary

The machine code that actually runs on computers isn't designed for humans to read and understand. To be usable, it needs to be converted into a different form.

One option for this is decompilation, which produces a result that is similar or identical to the original source code. However, decompilation is not always possible.

For fully compiled languages, such as C/C++ , and many other languages, it is necessary to disassemble a compiled executable and analyze it in assembly. However, this requires a much deeper understanding of the computer's architecture and how it actually works than writing and reading code in a higher-level language. Now that we know the role decompilation can play and the need for disassembly, in the next few chapters we'll look at how computers work, so we can learn to disassemble like a pro.