

Prologue

This paper is intended as an introduction to reverse engineering for someone who has no experience on the subject. You should have some basic knowledge of C programming, and access to a Windows or Linux box (preferably both) using the x86 architecture (i.e., your average computer). No knowledge of assembly code, registers, or the like is assumed, although it helps.

This introduction section of the paper is intended for the newcomer who has little or no understanding of what reverse engineering is, and may be skipped by those looking for more technical details.

What is Reverse Code Engineering?

"Reverse engineering (RE) is the process of discovering the technological principles of a mechanical application through analysis of its structure, function and operation"(Wikipedia). Basically, Reverse Code Engineering (RCE) is the application of the reverse engineering process to software - in other words, analyzing a program in order to understand how it works. Because reverse engineering is most commonly used to analyze closed-source programs, it is largely focused on the Windows platform; however, reversing under Linux is also popular for inspecting buffer overflows, closed-source Linux applications, and hostile Windows programs (without the risk of running them).

Why Reverse Engineer?

There are many reasons to reverse engineer a program. Have you ever wished that your favorite Windows program had xyz functionality? Want to dissect malware or viruses? Look for and analyze a buffer overflow? Figure out how that hardware driver works so you can write one for Linux? Maybe you're just curious how a particular program works, but you don't have access to the source code? All of these are common reasons for reverse engineering an application, and as such, there are many varied facets of RCE that one may choose to focus on, each of which can take a substantial amount of knowledge and experience to become an expert in. This paper will give you with the basic knowledge to get started in RCE, providing a base to launch into which ever specialties you prefer.

How Does It Work?

This all sounds great, but how do we analyze a program for which we have no code? There are many ways to observe how a program interacts with the rest of your system, such as file and registry access (which can be helpful when reverse engineering), but these techniques still leave you with a black box - you don't know what is going on under the hood. In order to understand how we can analyze the internal workings of a program, some understanding of the compilation process is needed. When you compile your source code, there are three major steps that occur: translation of the source code into assembly code, assembly, and linking.

First, the source code is translated into assembly code by the compiler. Assembly is a very low-level programming language; it is composed of many simple instructions which deal directly with memory addresses and CPU registers. For instance, if you assign the number 1 to an integer variable in your source code, the resulting assembly code may look something like:

```
mov 0xffffffffb4,0x1
```

which moves the number 1 into 0xfffffb4, the memory address assigned to that particular variable. No matter what programming language you are using (C/C++, Delphi, VB, etc), all compiled languages must be first translated into assembly before being converted into the final binary program.

Next, an assembler translates the assembly code into machine-readable code; there is (usually) a one-to-one translation between the assembly and machine code. The final stage is performed by a linker, whose job it is to add in any library functions required by the program. The final result is a file that contains binary instructions which can be executed by the processor.

The point of all this is that since all programs are translated into assembly code, and assembly code can be translated directly into binary 1s and 0s, we can translate any binary program back into its assembly code through the aptly named process of disassembly. If you understand assembly code, you can follow the instructions to understand what the program is doing, and even translate it into a higher-level language such as C. Note that some languages can be automatically translated directly back into their original source code, or decompiled. While this process works well for some languages, it is generally very complex and imprecise for most programming languages, particularly C/C++. I encourage you to look into some of the ongoing decompiler projects, however, this paper will be focused only on disassembly.

Opposition to RCE

It is important to realize that for various reasons, people may not want you to reverse engineer their programs, and as such, they may implement encryption or advanced protection techniques which make it extremely hard to analyze the original assembly code. We will certainly not be covering these techniques in this paper, but it is good to keep in mind if you come across a disassembled program that doesn't seem to make any sense.

A second issue is the legality of RCE. Many EULAs prohibit reverse engineering, but this still may not make it necessarily illegal; like many digital laws, it is still somewhat undefined. However, I will quote the following from Exploiting Software:

These agreements [EULAs] usually contain language that strictly prohibits reverse engineering. However, these agreements may or may not hold up in court [Kaner and Pels, 1998].

The Uniform Computer Information Transactions Act (UCITA) poses strong restrictions on reverse engineering and may be used to help "click through" EULA's stand-up in court. Some states have adopted the UCITA (Maryland and Virginia as of this writing [February 2004]), which strongly affects your ability to reverse engineer legally.

Normally, there is no need to fear RE-restrictive laws, unless you plan to publicize your work. One exception would be cracking, or using reverse engineering to circumvent an application's registration scheme, which is very illegal. All programs we will be working with in this paper are original, so there is no question of legality; however, it is very important to keep this in mind if you begin work on someone else's programs.

What Do I Need?

In short, tools and knowledge. Obviously, you must be able to read assembly code, however, it is not enough to just understand assembly instructions. You must also know how assembly instructions interact with areas of memory (particularly the stack), and what the CPU registers are used for. Knowledge of the high-level programming language that the application was written in can be very helpful, although it is not necessary. You should also understand specific system functions for the OS platform you are dealing with (such as Linux syscalls or the Windows API).

There are many tools available to the reverse engineer, much of them designed for specific purposes. However, there are two indispensable tools: the disassembler and the debugger. As its name implies, a disassembler disassembles a program's binary 1s and 0s into readable assembly code. A debugger can disassemble the binary instructions as well, but also allows you to run the code inside of the debugger; this gives you the distinct advantage of being able to observe the effect each instruction in real time, and allows you to better understand the program flow. The most popular debugger for Linux is the GNU debugger (gdb), which is also available for Windows; however, there are other very powerful debuggers for the Windows platform as well, such as SoftIce and OllyDbg. We will be using gdb in both Linux and Windows later in this paper.