Software Reverse Engineering

Jacco Krijnen

June 19, 2013

Abstract

While reverse engineering probably started with the analysis of hardware, today it plays a significant role in the software world. We discuss some of its uses and explain security related issues like malicious software and software piracy. It is shown how tools such as disassemblers and decompilers can help in this process, and why it is difficult to completely protect software from being reverse engineered.

1 Introduction

Trying to understand the world around us is part of what makes us human. It is the reason that we build particle accelerators, travel in space, and do scientific research in general. We like to take things apart and put them back together. Reverse engineering means working backwards from something man made, to gain knowledge about the internal workings or underlying design.

Whereas software engineering can be seen as implementing functional and technical specifications, software reverse engineering is the act of taking a piece of software (the implementation) and reconstructing original specifications or design. From now on we will refer to this simply as reverse engineering.

Reverse engineering is tightly related with security, some topics include:

- Detecting and analyzing malicious software
- Testing encryption systems
- Locating software vulnerabilities

These topics, among other uses, will be discussed in section two.

Since programs in their binary form are highly unreadable to most people, we would like to translate them in a higher level language and reverse as much as we can from what the compiler did. Techniques to do this will be discussed in section three. Another obvious question is: “Is there a way to protect my software from being reverse engineered?” This question, and more on protecting software will be discussed in section four.
2 Uses

2.1 Malicious software

The detection and understanding of malicious software (or malware) can be divided into two techniques: static and dynamic analysis. Dynamic analysis tries to analyze the program while it is running, by closely monitoring its behaviour with for instance a debugger. With static analysis, the code is reverse engineered and studied. Both have their strengths and weaknesses. For instance, advanced malware might notice that it is being observed and behave differently, defeating forms of dynamic analysis. On the other hand, code can be obfuscated (see section 4.1) or packed (which means encrypted or compressed) to hide as much as possible.

We can also look at it from a completely different angle, namely the perspective of the malware developer. It can use reverse engineering to locate vulnerabilities in an OS, or a protocol, in order to spread the malware. Sometimes, a malware developer might be interested in personal information which is stored in other software. In that case, the malware could be built to exploit a certain vulnerability in such software.

2.2 Encryption systems

Kerckhoffs’s principle\(^1\) states that we should not try to keep the encryption algorithm secret. If an enemy would ever get their hands on the software, it would be a matter of time before it is reverse engineered, which essentially breaks the algorithm.

So let us assume that the algorithms conform to Kerckhoffs’s principle. In most well designed algorithms, the best option to decrypt an intercepted message would be to bruteforce all the keys, so that appears to be a dead end. However, we should still try to reverse engineer private implementations of the algorithm. A single mistake in an implementation could create potential loopholes.

2.3 Developing competing software

2.3.1 Clean room design

A famous reverse engineering example took place during the 1980s, when Columbia Data Products developed a PC that was very similar to the IBM PCs. Since copying the proprietary BIOS software would have been illegal, a similar product was developed using the so called clean room design or Chinese wall. The method works as following

- A team examines the original software and comes up with a description of its functional specifications

- The description is implemented by a different team that has no connections with the specification team whatsoever

\(^1\)The security of an encryption system should only depend on the secrecy of the key, not the system itself.
The second team therefore has no knowledge of the actual techniques used in the product that is being reverse engineered. It should be noted that this method is a way to avoid copyright problems, since it is based on the principle of independent invention. Patents, however, do offer protection to the clean room approach [1].

Clean room approach (source: http://c2.com/cgi/wiki?ReverseEngineering)

3 Tools and Techniques

The reverse engineering process starts with an executable program. This can either be native machine code (for instance a windows .exe file) or code for a virtual machine (for instance a java .class file). Since programs in this binary format are highly unreadable to most people, we need a layer of abstraction.

3.1 Machine code and assembly language

Every processor architecture has an instructionset; a collection of operation codes, or opcodes which are represented by a fixed amount of bits. These are the elementary operations that a processor can execute. Opcodes can take arguments, which are called operands. For example, the instruction to pop 32-bits from the stack on an x86 architecture is 00001111, or simply 0F in hexadecimal notation [2].

An assembly language is a language of which the semantics have a one-to-one correspondence with a specific instructionset. The main reason is that it makes machine code readable. This is done by giving every instruction a name, a so called mnemonic. For example, the mnemonic would be pop. An assembler is a tool that performs this mapping; it converts the mnemonics to their associated opcodes. Assembly languages are often classified as low level programming languages, since it is the closest that you can get to the processor as programmer.
3.2 Disassemblers

To map machine code to assembly language, the opposite of an assembler is used, called a disassembler. The process itself is not exactly difficult, it could be implemented by linearly going through the binary opcodes, and retrieving the mnemonics with table lookups. Difficulties arise when trying to distinguish the actual code from data, since these two are combad. A simple linear scanning-and-translating algorithm therefore does not suffice, and some form of control flow analysis has to be performed. For a more advanced algorithm, see [3]

Also keep in mind that executables may have different file formats. The two most common types being PE (Portable Executable, Windows) and ELF (Executable and Linking Format, Unix). Both contain file headers that describe the entry point of the application (the address of the first instruction)[4][5].

3.3 Compilers

Most software is written in high level languages, such as C++ or Java. A compiler translates programs in these languages to machine code. This is a complex process, with some aspects to keep in mind:

• Comments and documentation are removed, since these have no meaning to the processor.

• Often, a compiler will perform various transformations to the original program for performance reasons, before generating the machine code.

• In the final machine code, identifiers (such as variable and function names) are omitted

This should make it clear that compiling is a lossy process. To be more precise, it is a many-to-many process. Source code can be compiled in different ways, with different optimizations, and machine code can be translated back in many ways.

3.4 Decompilers

While disassemblers can help us to get low level assembly language from machine language, we are still looking at the lowest level of abstraction. Stack operations, registers and pointers are all very close to the machine. To get the big picture, we would like a representation of the same program in a high level language. Because of the reasons mentioned above, it is nearly always impossible to obtain a copy of the original source code. However, it is possible to obtain a reasonable representation of it.

3.4.1 Malware revisited

The increased popularity of smartphones and tablets has not gone unnoticed by malware developers [6]. The static analysis and detection of malware, as discussed in section 2.1, is based on disassemblers and decompilers. This causes a problem for the disassembler, since different smartdevice companies use different
processor architectures, which have different instruction sets. In [7], a decompiler is presented that solves this problem. Its disassembler is retargetable, effectively making it platform independent.

4 Protecting software

Although traditionally, malicious software (the so called “client”) attacks a benign operating system (the “host”), it can also be the other way round, in which a benign client is attacked by a malicious host. This view is nicely and with more detail explained in [8]. Attacks include software piracy and malicious reverse engineering.

Protection against a malicious client is much easier than protecting against a malicious host, since a host can restrict the actions of a malicious client. In contrast, once a malicious host has his hands on a client, it can use any technique to try and attack it.

4.1 Reverse engineering and code obfuscation

Many believe that it is impossible to completely protect software against reverse engineering [9]. Still, there are many popular techniques that might make the process a lot harder, such as watermarking, temper. We discuss a popular techniques called code obfuscation.

Intuitively, code obfuscation is the process of transforming a program, such that its observable behaviour stays the same. This notion is formalized in [10]. You could say that an optimizing compiler already creates some obfuscation, but for different purposes: “These [obfuscations] are typically non-platform-specific transformations that modify the code to hide its original purpose and drown the reverser in a sea of irrelevant information”[3]. Code obfuscation therefore falls in the category of “security through obscurity” and “Security through obesity”. Code obfuscation often comes at the cost of performance.

4.1.1 Opaque predicates

A key ingredient is an opaque predicate: a boolean expressions that has to be evaluated at runtime, but of which the outcome is a constant value and known apriori. They can be used to introduce branches (for instance an if-then-else construction).[11] generalizes the notion of an opaque predicate to an opaque expression. This makes it possible to use a technique known as jump table spoofing.

4.1.2 Jump tables

A jump table is an array of pointers that point to possible execution branches. A compiler might implement branching (such as if-then-else or switch constructs) with jump tables. The idea of jump table spoofing is to take an ordinary, unconditional jump (for instance a function call) to address x and tranforming it into a jump table that contains arbitrary pointers into the program code (also called “junk addresses”), and also includes x. By using an opaque expression that evaluates to the index of x in the table, the program flow remains unchanged. For a reverse engineer, it might be very hard to understand the
program flow this way.

More advanced techniques of code obfuscation can be found in [3]

4.2 Software Piracy

A different type of software protection is the battle against software piracy. Digital content is extremely easy to move around and to copy, since the smallest building block are bits, and all hardware and software is built around this representation. Compare this to the middle ages, when monks had to work years, if not a lifetime, to make a single copy of the bible. With the internet, sharing has become instant, no need to hand over CD-ROMs. Copyright owners now face the reality that they have almost zero control over what happens with their digital material. A direct result is an increased amount of software pirating. Pirating software means the unauthorized usage, copying or redistributing of software.

4.3 Protecting against software piracy

A common protection is the use of serial numbers. These keys come separately with the software, and are validated by the software using a secret algorithm. Of course, this approach has a few major flaws: keys can be shared by users, and once the secret algorithm is reverse engineered, key generators can be built. A key generator is a small piece of software that has the knowledge of the validation algorithm and can provide a user with a valid key.

A famous example is the first Starcraft game, which had 13-digit keys that relied on such an algorithm. People noticed that trying some keys would often work. Eventually, the algorithm was reverse engineered and discovered that it relied on a simple checksum where the thirteenth bit verifies the first twelve.

An improvement upon serial numbers is the use of online activation. Now, the software vendor is able to tell if the serial number is legitimate. Still, it is possible to crack the software and setup a key generator that performs a man in the middle attack [3]. Another drawback is the fact that the customer has to rely on everlasting support from the publisher, what if the publisher go out of business?

A completely different approach is Hardware based protection, usually this is in the form of a USB dongle. The program installer can then check if the dongle is present in a USB port. A simple check however, does not suffice, it is relatively easy to crack since it relies on interoperability with the OS. A better solution would be to encrypt the actual program with a key, which is present on the dongle.

References


