An Introduction to Software Reverse Engineering

Teodoro Cipresso, <tcipress@hotmail.com>
San Jose State University, Silicon Valley
IBM, Silicon Valley

Abstract
Software Reverse Engineering (SRE) is the practice of analyzing a software system, either in whole or in part, to extract design and implementation information. A typical SRE scenario would involve a software module that has worked for years and carries several rules of a business in its lines of code; unfortunately the source code of the application has been lost—what remains is “native” or “binary” code. Reverse engineering skills are also used to detect and neutralize viruses and malware, and to protect intellectual property. Computer programmers proficient in software reverse engineering will be needed should software components like these need to be maintained, enhanced, or reused. It became frightfully apparent during the Y2K crisis that reverse engineering skills were not commonly held amongst programmers. Since that time, much research has been underway to formalize just what types of activities fall into the category of reverse engineering, so that these skills could be taught computer programmers and testers. To help address the lack of software reverse engineering education, several peer-reviewed articles on software reverse engineering, re-engineering, reuse, maintenance, evolution, and security were gathered with the objective of developing relevant, practical exercises for instructional purposes. The research revealed that SRE is fairly well described and all related activities mostly fall into one of two categories: software development-related and security-related. Hands-on reversing exercises were developed in the spirit of these two categories with the goal of providing a baseline education in reversing both Wintel machine code and Java bytecode.

1 Why Learn About Software Reverse Engineering

From very early on in life we engage in constant investigation of existing things to understand how and even why they work. The practice of Software Reverse Engineering (SRE) calls upon this investigative nature when one needs to learn how and why, often in the absence of adequate documentation, an existing piece of software—helpful or malicious—works. In the sections that follow we cover the most popular uses of SRE and, to some degree, the importance of imparting knowledge of them to those who write, test, and maintain software. More formally, SRE can be described as the practice of analyzing a software system to create abstractions that identify the individual components and their dependencies, and, if possible, the overall system architecture [1], [2]. Once the components and design of an existing system have been recovered, it
becomes possible to repair and even enhance them.

Events in recent history have caused SRE to become a very active area of research. In the early nineties, the Y2K problem spurred the need for the development of tools that could read large amounts of source or binary code for the 2-digit year vulnerability [2]. Not too far after the Y2K problem, in the mid to late nineties, the adoption of the Internet by businesses and organizations brought about the need to understand in-house legacy systems so that the information held within them could be made available on the Web [3]. The desire for businesses to expand to the Internet for what was promised to be limitless potential for new revenue caused the creation of many B2C (Business to Consumer) Web sites.

Today’s technology is unfortunately tomorrow’s legacy system. For example, the Web 2.0 revolution sees the current crop of websites as legacy Web applications comprised of multiple HTML pages; Web 2.0 envisions sites where a user interacts with a single dynamic page—rendering a user experience that is more like traditional desktop applications [2]. Porting the current crop of legacy web sites to Web 2.0 will require understanding the architecture and design of these legacy sites—again requiring reverse engineering skills and tools.

At first glance it may seem that the need for SRE can be lessened by simply maintaining good documentation for all software that is written. While the presence of that ideal would definitely lower the need--it just has not become a reality. For example, even a company that has brought software to market may no longer understand it because the original designers and developers may have left, or components of the software have been acquired from a vendor—who could no longer be in business [1].

Going forward, the vision is to include SRE incrementally, as part of the normal development, or “forward engineering” of software systems. At regular points during the development cycle, code would be reversed to rediscover its design so that the documentation can be updated. This would help avoid the typical situation where detailed information about a software system such as its architecture, design constraints and trade-offs is found only in the memory of its developer [1].

2 Reverse Engineering in Software Development

While a great deal of software that has been written is no longer in use, a considerable amount has survived for decades and continues to run the global economy. The reality of the situation is that 70% of the source code in the entire world is written in COBOL [3]. One would be hard-pressed these days to obtain an expert education in legacy programming languages like COBOL, PL/I, and FORTRAN. Compounding the situation is the fact that a great deal of legacy code is poorly designed and documented [3]. [6] states that “COBOL programs are in use globally in governmental and military agencies, in commercial enterprises, and on operating systems such as IBM's z/OS®, Microsoft's Windows®, and the POSIX families (Unix/Linux etc.). In 1997, the Gartner Group reported that 80% of the world's business ran on COBOL with over 200 billion
lines of code in existence and with an estimated 5 billion lines of new code annually.” Since it’s cost-prohibitive to rip and replace billions of lines of legacy code, the only reasonable alternative has been to maintain and evolve the code, often with the help of concepts found in software reverse engineering. Figure 2.1 illustrates a process a software engineer might follow when maintaining legacy software systems.

![Figure 2.1 Development process for maintaining legacy software.](image)

Whenever computer scientists or software engineers are engaged with evolving an existing system, fifty to ninety percent of the work effort is spent on program understanding [3]. Having engineers spend such a large amount of their time attempting to understand a system before making enhancements is not economically sustainable as a software system continues to grow in size and complexity. To help lessen the cost of program understanding, [3] advises that “practice with reverse engineering techniques improves ability to understand a given system quickly and efficiently.”

Even though several tools already exist to aid software engineers with the program understanding process, the tools focus on transferring information about a software system’s design into the mind of the developer [1]. The expectation is that the developer has enough skill to efficiently integrate the information into their own mental model of the system’s architecture. It’s not likely that even the most sophisticated tools can replace experience with building mental model of existing software; [4] states “commercial reverse engineering tools produce various kinds of output, but software engineers usually don’t how to interpret and use these pictures and reports.” The lack of reverse engineering skills in most programmers is a serious risk to the long-term viability of any organization that employs information technology. The problem of software maintenance cannot be dispelled with some clever technique, [7] argues “re-engineering code to create a system that will not need to be reverse engineered again in the future—is presently unattainable.”
According to [5], there are four software development-related reverse engineering scenarios; the scenarios cover a broad spectrum of activities that include software maintenance, reuse, re-engineering, evolution, interoperability, and testing. Figure 2.2 summarizes the software development-related reverse engineering scenarios.

The following are tasks one might perform in each of the reversing scenarios [5]:

- **Achieving Interoperability with Proprietary Software**: Develop applications or device drivers that interoperate (use) proprietary libraries in operating systems or applications.
- **Verification that Implementation Matches Design**: Verify that code produced during the forward development process matches the envisioned design by reversing the code back into an abstract design.
- **Evaluating Software Quality and Robustness**: Ensure the quality of software before purchasing it by performing heuristic analysis of the binaries to check for certain instruction sequences that appear in poor quality code.
- **Legacy Software Maintenance, Re-engineering, and Evolution**: Recover the design of legacy software modules when source is not available to make possible the maintenance, evolution, and reuse of the modules.
3 Reverse Engineering in Software Security

From the perspective of a software company, it is highly desirable that the company’s products are difficult to pirate and reverse engineer. Making software difficult to reverse engineer seems to be in conflict with the idea of being able to recover the software’s design later on for maintenance and evolution. Therefore, software manufacturers usually don’t apply anti-reverse engineering techniques to software until it is shipped to customers, keeping copies of the readable and maintainable code. Software manufacturers will typically only invest time in making software difficult to reverse engineer if there are particularly interesting algorithms that make the product stand out from the competition.

Making software difficult to pirate or reverse engineer is often a moving target and requires special skills and understanding on the part of the developer. Software developers who are given the opportunity to practice antireversing techniques might be in a better position to help their employer, or themselves, protect their intellectual property. As [3] states, “to defeat a crook you have to think like one.” By reverse engineering viruses or other malicious software, programmers can learn their inner workings and witness first-hand how vulnerabilities find their way into computer programs. Reversing software that has been infected with a virus, is a technique used by the developers of antivirus products to identify and neutralize new viruses or understand the behavior of malware.

Programming languages like Java, which do not require computer programmers to manage low-level system details, have become ubiquitous. As a result, computer programmers have increasingly lost touch with what happens in a system during execution of programs. [3] suggests that programmers can gain a better and deeper understanding of software and hardware through learning reverse engineering concepts. Hackers and crackers have been quite vocal and active in proving that they possess a deeper understanding of low-level system details than their professional counterparts [3].

According to [5], there are four software security-related reverse engineering scenarios; just like development-related reverse engineering—the scenarios cover a broad spectrum of activities that include: ensuring that software is safe to deploy and use, protecting clever algorithms or business processes, preventing pirating of software and digital media such as music, movies, and books—and making sure that cryptographic algorithms are not vulnerable to attacks. Figure 3.1 summarizes the software security-related reverse engineering scenarios. The following are tasks one might perform in each of the reversing scenarios [5]:

- **Detecting and Neutralizing Viruses and Malware**: Detect, analyze, or neutralize (clean) malware, viruses, spyware, and adware.
- **Testing Cryptographic Algorithms for Weaknesses**: Test the level of data security provided by a given cryptographic algorithm by analyzing it for weaknesses.
• **Testing DRM or License Protection (anti-reversing):** Protect software and media digital-rights through application and testing of antireversing techniques.

• **Auditing the Security of Program Binaries:** Audit a program for security vulnerabilities without access to the source code by scanning instruction sequences for potential exploits.

![Figure 3.1. Security-related software reverse engineering scenarios.](image)

### 4 Reversing and Patching Wintel Machine Code

The executable representation of software, otherwise known as machine code, is typically the result of translating a program written in a high-level language, using a compiler, to an object file, a file which contains platform-specific machine instructions. The object file is made executable using linker, a tool which resolves the external dependencies that the object file has, such as operating system libraries. In contrast to high-level languages, there are low-level languages which are still considered to be high-level by a computer's CPU because the language syntax is still a textual or mnemonic abstraction of the processor's instruction set. For example, assembly language, a language that uses helpful mnemonics to represent machine instructions, still must be translated to an object file and made executable by a linker. However the translation from assembly code to machine code is done by an assembler instead of a compiler—reflecting the closeness of the assembly language's syntax to actual machine code.

The reason why compilers translate programs coded in high-level and low-level languages to machine code is three-fold: CPUs only understand machine instructions, having a CPU dynamically translate higher-level language statements to machine instructions would consume significant, additional CPU time, and (3) a CPU that could dynamically translate multiple high-level languages to machine code would be extremely complex, expensive, and cumbersome to maintain—imagine having to update the
firmware in your microprocessor every time a bug is fixed or a feature is added to the C++ language!

To relieve a high-level language compiler from the difficult task of generating machine instructions, some compilers do not generate machine code directly, instead they generate code in a low-level language such as assembly [8]. This allows for a separation of concerns where the compiler doesn't have to know how to encode and format machine instructions for every target platform or processor—it can instead just concentrate on generating valid assembly code for an assembler on the target platform. Some compilers, such as the C and C++ compilers in the GNU Compiler Collection (GCC), have the option to output the intermediate assembly code that the compiler would otherwise feed to the assembler—allowing advanced programmers to tweak the code [9]. Therefore the C and C++ compilers in GCC are examples of compilers that translate high-level language programs to assembly code instead of machine code; they rely on an assembler to translate their output into instructions the target processor can understand. [9] outlines the compilation process undertaken by GCC compiler to render an executable file is as follows:

- **Preprocessing**: Expand macros in the high-level language source file.
- **Compilation**: Translate the high-level source code to assembly language.
- **Assembly**: Translate assembly language to object code (machine code).
- **Linking** (Create the final executable):
  - Statically or dynamically link together the object code with the object code of the programs and libraries it depends on.
  - Establish initial relative addresses for the variables, constants, and entry points in the object code.

### 4.1 Decompilation and Disassembly of Machine Code

Having an understanding of how high-level language programs become executable machine code can be extremely helpful when attempting to reverse engineer one. Most software tools that assist in reversing executables work by translating the machine code back into assembly language. This is possible because there exists a one-to-one mapping from each assembly language instruction to a machine instruction [10]. A tool that translates machine code back into assembly language is called a disassembler.

From a reverse engineer's perspective the next obvious step would be to translate assembly language back to a high-level language, where it would be much less difficult to read, understand, and alter the program. Unfortunately, this is an extremely difficult task for any tool because once high-level language source code is compiled down to machine code, a great deal of information is lost. For example, one cannot tell by looking at the machine code which high-level language (if any) the machine code originated from. Perhaps knowing a particular quirk about a compiler might help a reverse engineer identify some machine code that it had a hand in creating, but this is not a reliable
The greatest difficulty in reverse engineering machine code comes from the lack of adequate decompilers—tools that can generate equivalent high-level language source code from machine code. The paper [5] argues that it should be possible to create good decompilers for binary executables, but recognizes that other experts disagree—raising the point that some information is “irretrievably lost during the compilation process.” Boomerang is a well-known open-source decompiler project that seeks to one day be able to decompile machine code to high-level language source code with respectable results [11]. For those reverse engineers interested in recovering the source code of a program, decompilation may not offer much hope because as [11] states “a general decompiler does not attempt to reverse every action of the compiler, rather it transforms the input program repeatedly until the result is high level source code. It therefore won't recreate the original source file; probably nothing like it.”

To get a sense of the effectiveness of Boomerang as a reversing tool, a simple program, *HelloWorld.c* was compiled and linked using the GNU C++ compiler for Microsoft Windows® and then decompiled using Boomerang. The C code generated by the Boomerang decompiler when given *HelloWorld.exe* as input was quite disappointing: the generated code looked like a hybrid of C and assembly language, had countless syntax errors, and ultimately bore no resemblance to the original program. Table 4.1 contains the source of *HelloWorld.c* and some of the code generated by Boomerang. Incidentally, the Boomerang decompiler was unable to produce any output when *HelloWorld.exe*, was built using Microsoft’s Visual C++ 2008 edition compiler.

**Table 4.1. Result of decompiling *HelloWorld.exe* using Boomerang.**

HelloWorld.c:

```c
01: #include <stdio.h>
02: int main(int argc, char *argv[])
03: {
04:   printf("Hello Boomerang World\n");
05:   return 0;
06: }
```

Boomerang decompilation of HelloWorld.exe (abbreviated):

```c
01: union { __size32[] x83; unsigned int x84; } global10;
02: __size32 global3 = -1; // 4 bytes
03: 
04: // address: 0x401280
05: void _start()
06: {
07:   __set_app_type();
08:   procl();
09: }
10:
```
The full length of the C code generated by Boomerang for the HelloWorld.exe program contained 180 lines of confusing, nonsensical control structures and function calls to undefined methods. It is surprising to see such a poor decompilation result, but as [11] states: “Machine code decompilation, unlike Java/.NET decompilation, is still a very immature technology.” To ensure that decompilation was given a fair trial, another decompiler was tried on the HelloWorld.exe executable. The Reversing Engineering Compiler or REC is both a compiler and a decompiler that claims to be able to produce a “C-like” representation of machine code [12]. Unfortunately, the results of the decompilation using REC were similar to that of Boomerang. Based on the current state of decompilation technology for machine code, using a decompiler to recover the high-level language source of an executable doesn’t seem feasible; however, because of the one-to-one correspondence between machine code and assembly language statements [10], we can obtain a low-level language representation. Fortunately there are graphical tools available that not only include a disassembler, a tool which generates assembly language from machine code, but also allow for debugging and altering the machine code during execution.

4.2 Wintel Machine Code Reversing and Patching Exercise

Imagine that we have just implemented a C/C++ version of a Windows® 32-bit console application called “Password Vault” that helps computer users create and manage their passwords in a secure and convenient way. Before releasing a limited trial version of the application on our company’s Web site, we would like to understand how difficult it would be for a reverse engineer to circumvent a limitation in the trial version that exists to encourage purchases of the full version; the trial version of the application limits the number of password records a user may create to five.

The C++ version of the Password Vault application (included with this text) was developed to provide a non-trivial application for reversing exercises without the myriad of legal concerns involved with reverse engineering software owned by others. The Password Vault application employs 256-bit AES encryption, using the free
cryptographic library crypto++ [17], to securely store passwords for multiple users—each in separate, encrypted XML files. By default, the Makefile that is used to build the Password Vault application defines a constant named “TRIALVERSION” which causes the resulting executable to limit the number of password records a user may create to only five, using conditional compilation. This limitation is very similar to limitations found in many shareware and trialware applications that are available on the Internet.

4.3 Recommended Reversing Tool for the Wintel Exercise

OllyDbg is a shareware interactive machine code debugger and disassembler for Microsoft Windows® [13]. The tool has an emphasis on machine code analysis which makes it particularly helpful in cases where the source code for the target program is unavailable [13]. Figure 4.1 illustrates the OllyDbg graphical workbench. OllyDbg operates as follows: the tool will disassemble a binary executable, generate assembly language instructions from machine code instructions, and perform some heuristic analysis to identify individual functions (methods) and loops. OllyDbg can open an executable directly, or attach to one that is already running. The OllyDbg workbench can display several different windows which are made visible by selecting them on the View menu bar item. The CPU window, shown in Figure 4.1, is the default window that is displayed when the OllyDbg workbench is started. Table 4.2 lists the panes of the CPU window along with their respective capabilities; the contents of the table are adapted from the online documentation provided by [13] and experience with the tool.

Figure 4.1. The five panes of the OllyDbg graphical workbench.
<table>
<thead>
<tr>
<th>Pane</th>
<th>Capabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disassembler</td>
<td>➢ Edit, debug, test, and patch a binary executable using actions available on a popup menu.</td>
</tr>
<tr>
<td></td>
<td>➢ Patch an executable by copying edits to the disassembly back to the binary.</td>
</tr>
<tr>
<td>Dump</td>
<td>➢ Display the contents of memory or a file in one of 7 predefined formats: byte, text, integer, float, address, disassembly, or PE Header.</td>
</tr>
<tr>
<td></td>
<td>➢ Set memory breakpoints (triggered when a particular memory location is read from or written to).</td>
</tr>
<tr>
<td></td>
<td>➢ Locate references to data in the disassembly (executable code).</td>
</tr>
<tr>
<td>Information</td>
<td>➢ Decode and resolve the arguments of the currently selected assembly instruction in the Disassembler pane.</td>
</tr>
<tr>
<td></td>
<td>➢ Modify the value of register arguments.</td>
</tr>
<tr>
<td></td>
<td>➢ View memory locations referenced by each argument in either the Disassembler of Dump panes.</td>
</tr>
<tr>
<td>Registers</td>
<td>➢ Decodes and displays the values of the CPU and FPU (Floating-Point Unit) registers for the currently executing thread.</td>
</tr>
<tr>
<td></td>
<td>➢ Floating point register decoding can be configured for MMX (Intel) or 3DNow! (AMD) multimedia extensions.</td>
</tr>
<tr>
<td></td>
<td>➢ Modify the value of CPU registers.</td>
</tr>
<tr>
<td>Stack</td>
<td>➢ Display the stack of the currently executing thread.</td>
</tr>
<tr>
<td></td>
<td>➢ Trace stack frames. In general, stack frames are used to:</td>
</tr>
<tr>
<td></td>
<td>• Restore the state of registers and memory on return from a call statement.</td>
</tr>
<tr>
<td></td>
<td>• Allocate storage for the local variables, parameters, and return value of the called subroutine.</td>
</tr>
<tr>
<td></td>
<td>• Provide a return address.</td>
</tr>
</tbody>
</table>
4.4 Animated Solution to the Wintel Reversing Exercise

Using OllyDbg, one can successfully reverse engineer a non-trivial Windows® application like Password Vault, and make permanent changes to the behavior of the executable. The purpose of placing a trial limitation in the Password Vault application is to provide a concrete objective for reverse engineering the application: disable or relax the trial limitation. Of course the goal here is not teach how to avoid paying for software, but rather to see oneself in the role of a tester, a tester who is evaluating how difficult it would be for reverse engineer to circumvent the trial limitation. This is a fairly relevant exercise to go through for any individual or software company that plans to provide trial versions of their software for download on the Internet. In later sections, we discuss anti-reversing techniques, which can significantly increase the difficulty a reverse engineer will encounter when reversing an application.

For instructional purposes, an animated tutorial that demonstrates the complete end-to-end reverse engineering of the C++ Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Password Vault application and OllyDbg already installed on a Windows® XP machine. Figure 4.2 contains an example slide from the animated tutorial. The animated tutorial, source, and installer for the machine code version of Password Vault can be downloaded from the following locations:

- **Wintel Reversing & Patching Animated Solution:**
  [http://reversingproject.info/repository.php?fileID=4_1_1](http://reversingproject.info/repository.php?fileID=4_1_1)

- **Password Vault C/C++ Source code:**
  [http://reversingproject.info/repository.php?fileID=4_1_2](http://reversingproject.info/repository.php?fileID=4_1_2)

- **Password Vault C/C++ Windows® installer:**
  [http://reversingproject.info/repository.php?fileID=4_1_3](http://reversingproject.info/repository.php?fileID=4_1_3)

Begin viewing the animated tutorial by extracting `password_vault_cpp_reversing_exercise.zip` to a local directory and either running `password_vault_cpp_reversing_exercise.exe` which should launch the standalone version of Macromedia Flash Player, or by opening the file `password_vault_cpp_reversing_exercise_viewlet_swf.html` in a Web browser.
Figure 4.2. Sample slide from the machine code reversing animated tutorial.
5 Reversing and Patching Java Bytecode

Applications written in Java are generally well-suited to being reverse engineered. To understand why, it’s important to understand the difference between machine code and Java bytecode (Figure 5.1 illustrates the execution of Java bytecode versus machine code):

- **Machine code**: “Machine code or machine language is a system of instructions and data executed directly by a computer's central processing unit” [14]. Machine code contains the platform-specific machine instructions to execute on the target processor.

- **Java bytecode**: “Bytecode is the intermediate representation of Java programs just as assembler is the intermediate representation of C or C++ programs” [15]. Java bytecode contains platform-independent instructions that are translated to platform-specific instructions by a Java Virtual Machine.

In section 4, an attempt to recover the source of a simple “Hello World” C++ application was unsuccessful when the output of two different compilers was given as input to the Boomerang decompiler. Much more positive results can be achieved for Java bytecode because of its platform-independent design and high-level representation. On Windows®, machine code is typically stored in files with the extensions *.exe, *.dll; the file extensions for machine code vary per operating system. This is not the case with Java bytecode as it is always stored in files that have a *.class extension. Related Java classes, such as those for an application or class library, are often bundled together in an archive file with a *.jar extension. The Java Language Specification allows at most one top-level public class to be defined per *.java source file and requires that the bytecode be stored in a file with whose name matches TopLevelClassName.class.

**Figure 5.1.** Execution of Java bytecode versus machine code.
5.1 Decompiling and Disassembling Java Bytecode

To demonstrate how much more feasible it is to recover Java source code from Java bytecode than it is to recover C++ code from machine code, we decompile the bytecode for the program `ListArguments.java` using Jad, a Java decompiler which can be found here [16]; we then compare the generated Java source with the original. Before performing the decompilation we peek at the bytecode using `javap` to get an idea of how much information survives the translation from high-level Java source to the intermediate format of Java bytecode. Table 5.1 contains the source code for `ListArguments.java`, a simple Java program that echoes each argument passed on the command-line to standard output.

**Table 5.1.** Source listing for `ListArguments.java`.

```
01: package info.reversingproject.listarguments;
02: public class ListArguments {
03:   public static void main(String[] arguments){
04:     for (int i = 0; i < arguments.length; i++) {
05:       System.out.println("Argument[" + i + "]:" + arguments[i]);
06:     }
07:   }
08: }
09: }
```

Bytecode is stored in a binary format that is not human-readable and therefore must be “disassembled” in order to be read. Recall that the result of disassembling machine code is assembly language that can be converted back into machine code using an assembler; unfortunately, the same does not hold for disassembling Java bytecode. Sun Microsystems’s Java Development Toolkit (JDK) comes with `javap` a command-line tool for disassembling Java bytecode; to say that `javap` “disassembles” bytecode is a bit of a misnomer since the output of `javap` is unstructured text which cannot be converted back into bytecode. The output of `javap` is nonetheless useful as a debugging and performance tuning aid since one can see which JVM instructions are generated from high-level Java language statements.

Table 5.2 lists the Java bytecode for the `main` method of `ListArguments` class; notice that the fully qualified name of each method invoked by the bytecode is preserved. It may seem curious that while `ListArguments.java` contains no references to the class `java.lang.StringBuilder`, there are many references to it in the bytecode; this is because the use of the “+” operator to concatenate strings is a convenience offered by the Java language that has no direct representation in bytecode. To perform the concatenation, the bytecode creates a new instance of the `StringBuilder` class and invokes its `append` method for each occurrence of the “+” operator in the original Java source code (there are three). A loss of information has indeed occurred, but we’ll see that it’s still possible to generate Java source code equivalent to the original in function, but not in syntax.
Table 5.2. Java bytecode contained in ListArguments.class.

<table>
<thead>
<tr>
<th>No.</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>iconst_0</td>
<td>Push 0 onto the stack.</td>
</tr>
<tr>
<td>1</td>
<td>istore_1</td>
<td>Store the result of the previous instruction.</td>
</tr>
<tr>
<td>2</td>
<td>iload_1</td>
<td>Load the result of the previous instruction.</td>
</tr>
<tr>
<td>3</td>
<td>aload_0</td>
<td>Load the value from the specified location.</td>
</tr>
<tr>
<td>4</td>
<td>arraylength</td>
<td>Get the array length.</td>
</tr>
<tr>
<td>5</td>
<td>if_icmpge 50</td>
<td>If the operand stack contains an integer, branch if greater than or equal to 50.</td>
</tr>
<tr>
<td>8</td>
<td>getstatic #2;</td>
<td>Get the static reference to System.out.</td>
</tr>
<tr>
<td>11</td>
<td>new #3;</td>
<td>Create a new StringBuilder instance.</td>
</tr>
<tr>
<td>14</td>
<td>dup</td>
<td>Duplicate the top of the operand stack.</td>
</tr>
<tr>
<td>15</td>
<td>invokespecial #4;</td>
<td>Invoke a special method on the StringBuilder.</td>
</tr>
<tr>
<td>18</td>
<td>ldc #5</td>
<td>Load an integer constant.</td>
</tr>
<tr>
<td>20</td>
<td>invokevirtual #6;</td>
<td>Invoke a virtual method on the StringBuilder.</td>
</tr>
<tr>
<td>23</td>
<td>iload_1</td>
<td>Load the result of the previous instruction.</td>
</tr>
<tr>
<td>24</td>
<td>invokevirtual #7;</td>
<td>Invoke a virtual method on the StringBuilder.</td>
</tr>
<tr>
<td>27</td>
<td>ldc #8</td>
<td>Load an integer constant.</td>
</tr>
<tr>
<td>29</td>
<td>invokevirtual #6;</td>
<td>Invoke a virtual method on the StringBuilder.</td>
</tr>
<tr>
<td>32</td>
<td>aload_0</td>
<td>Load the value from the specified location.</td>
</tr>
<tr>
<td>33</td>
<td>iload_1</td>
<td>Load the result of the previous instruction.</td>
</tr>
<tr>
<td>34</td>
<td>aaload</td>
<td>Load a single element from an array.</td>
</tr>
<tr>
<td>35</td>
<td>invokevirtual #6;</td>
<td>Invoke a virtual method on the StringBuilder.</td>
</tr>
<tr>
<td>38</td>
<td>invokevirtual #9;</td>
<td>Invoke a virtual method on the StringBuilder.</td>
</tr>
<tr>
<td>41</td>
<td>invokevirtual #10;</td>
<td>Invoke a virtual method on the PrintStream.</td>
</tr>
<tr>
<td>44</td>
<td>iinc 1</td>
<td>Increment the operand stack index by 1.</td>
</tr>
<tr>
<td>47</td>
<td>goto 2</td>
<td>Jump to the specified label.</td>
</tr>
<tr>
<td>50</td>
<td>return</td>
<td>Return from the method.</td>
</tr>
</tbody>
</table>

Table 5.3 lists the result of decompiling ListArguments.class using Jad; while the code is different from the original ListArguments.java program, it is functionally equivalent and syntactically correct, which is a much better result than that seen earlier with decompiling machine code.

Table 5.3. Jad decompilation of ListArguments.class.

```
01: package info.reversingproject.listarguments;
02: import java.io.PrintStream;
03:
04: public class ListArguments
05: {
06:   public static void main(String args[])
07:   {
08:     for (int i = 0; i < args.length; i++)
09:       System.out.println((new StringBuilder()).append("Argument[")
10:         .append(i).append("]:").append(args[i]).toString());
11:   }
12: }
```
An advanced programmer who is fluent in the Java Virtual Machine specification could use a hex editor or a program to modify Java bytecode directly, but this is similar to editing machine code directly, which is error-prone and difficult. In section 4, which covered reversing and patching of machine code, it was determined through discussion and an animated tutorial that one should work with disassembly to make changes to a binary executable. However, the result of disassembling Java bytecode is a pseudo-assembly language, a language that cannot be compiled or assembled but serves to provide a more abstract, readable representation of the bytecode. Being that editing bytecode directly is difficult, and that disassembling bytecode results in pseudo-assembly which cannot be compiled, it would at first seem that losing Java source code is more dire of a situation than losing C++ code, but of course this is not the case since, as we've seen using Jad, Java bytecode can be successfully decompiled to equivalent Java source code.

5.2 Java Bytecode Reversing and Patching Exercise

This section introduces an exercise that is the Java Bytecode equivalent of that given in Section 4.2 for Wintel machine code. Imagine that we have just implemented a Java version of the a console application called “Password Vault” that helps computer users create and manage their passwords in a secure and convenient way. Before releasing a limited trial version of the application on our company’s Web site, we would like to understand how difficult it would be for a reverse engineer to circumvent a limitation in the trial version that exists to encourage purchases of the full version; the trial version of the application limits the number of password records a user may create to five.

The Java version of the Password Vault application (included with this text) was developed to provide a non-trivial application for reversing exercises without the myriad of legal concerns involved with reverse engineering software owned by others. The Java version of the Password Vault application employs 128-bit AES encryption, using Sun's Java Cryptography Extensions (JCE), to securely store passwords for multiple users—each in separate, encrypted XML files.

5.3 Recommended Reversing Tool for the Java Exercise

If using Jad from the command-line doesn't sound appealing there is a freeware graphical tool built upon Jad called FrontEnd Plus that provides a simple workbench for decompiling classes and browsing the results [16]; it also has a convenient batch mode where multiple Java class files can be decompiled at once. After editing the Java generated by Jad, it’s necessary to recompile the source back to bytecode in order to integrate the changes. The ability to recompile the generated Java is not functional in the FrontEnd Plus workbench for some reason, though it’s simple enough to do the compilation manually. Next we mention an animated tutorial for reversing a Java implementation of the Password Vault application, which was introduced in section 4. Figure 5.2 shows a FrontEnd Plus workbench session containing the decompilation of
ListArguments.class.

To demonstrate using the FrontEnd Plus to reverse engineer and patch a Java bytecode, a Java version of the Password Vault application was developed; recall that the animated tutorial in section 4 introduced the machine code (C++) version. The Java version of the Password Vault application uses 128-bit instead of 256-bit AES encryption because Sun Microsystem's standard Java Runtime Environment (JRE) does not provide 256-bit encryption due to export controls. A trial limitation of five password records per users is also implemented in the Java version. Unfortunately, Java does not support conditional compilation, so the source code cannot be compiled to omit the trial limitation without manually removing it or using a custom build process.

Figure 5.2. FrontEnd Plus workbench session for ListArguments.class.
5.4 Animated Solution to the Java Reversing Exercise

Using FrontEnd Plus (and Jad), one can successfully reverse engineer a non-trivial Java application like Password Vault, and make permanent changes to the behavior of the bytecode. Again, the purpose of having placed a trial limitation in the Password Vault application is to provide an opportunity for one to observe how easy or difficult it is for a reverse engineer to disable the limitation. Just like for machine code, antireversing strategies can be applied to Java bytecode. We cover some basic, effective strategies for protecting bytecode from being reverse engineered in a later section.

For instructional purposes, an animated solution that demonstrates the complete end-to-end reverse engineering of the Java Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Java Password Vault application, FrontEnd Plus, and Sun's Java JDK v1.6 installed on a Windows XP® machine. Figure 5.3 contains an example slide from the animated tutorial. The animated tutorial, source, and installer for the Java version of Password Vault can be downloaded from the following locations:

- Java Bytecode Reversing & Patching Animated Solution:  
  http://reversingproject.info/repository.php?fileID=5_4_1
- Password Vault Java Source code:  
  http://reversingproject.info/repository.php?fileID=5_4_2
- Password Vault (Java Version) Windows® installer:  
  http://reversingproject.info/repository.php?fileID=5_4_3

Begin viewing the tutorial by extracting password_vault_java_reversing_exercise.zip to a local directory and either running password_vault_java_reversing_exercise.exe which should launch the standalone version of Macromedia Flash Player, or by opening the file password_vault_java_reversing_exercise_viewlet.swf.html in a Web browser.
6 Basic Antireversing Techniques

Having seen that it is fairly straight-forward for a reverse engineer to disable the trial limitation on the machine code and Java bytecode implementations of the Password Vault application, we now investigate applying antireversing techniques to both implementations in order to make it significantly more difficult for the trial limitation to be disabled. While antireversing techniques cannot completely prevent software from being reverse engineered, they act as a deterrent by increasing the challenge for the reverse engineer. [5] states “It is never possible to entirely prevent reversing” and “What is possible is to hinder and obstruct reversers by wearing them out and making the process so slow and painful that they give up.” The remainder of this section introduces basic antireversing techniques, two of which are demonstrated in sections 7 and 8.

While it is not possible to completely prevent software from being reverse engineered, a reasonable goal is to make it as difficult as possible. Implementing antireversing strategies for source code, machine code, and bytecode can have adverse effects on a program's size, efficiency, and maintainability; therefore, it’s important to evaluate whether a particular program warrants the cost of protecting it. The basic antireversing techniques introduced in this section are meant to be applied post-production, after the coding for an application is complete and tested. These techniques obscure data and logic and therefore are difficult to implement while also working on the actual functionality of the application — doing so could hinder or slow debugging and, even worse, create a dependency between the meaningful program logic and the antireversing strategies used. [5] describes three basic antireversing techniques:

- **Eliminating Symbolic Information:** The first and most obvious step in preventing reverse engineering of a program is to render unrecognizable, all symbolic information in machine code or bytecode because such information can be quite useful to a reverse engineer. Symbolic information includes class names, method names, variable names, and string constants that are still readable after a program has been compiled down to machine code or bytecode.

- **Obfuscating the Program:** Obfuscation includes eliminating symbolic information, but goes much further. Obfuscation strategies include: modifying the layout of a program, introducing confusing non-essential logic or control flow, and storing data in difficult to interpret organizations or formats. Applying all of these techniques can render a program difficult to reverse, however care must be taken to ensure the original functionality of the application remains intact.

- **Embedding Antidebugger Code:** Static analysis of machine code is usually carried out using a disassembler and heuristic algorithms that attempt to understand the structure of the program. Active or live analysis of machine code is done using an interactive debugger-disassembler that can attach to a running program and allow a reverse engineer to step through each instruction and observe the behavior of the program at key points during its execution. Live analysis is how most reverse
engineers get the job done, so it’s common for developers to want to implement guards against binary debuggers.

7 Applying Antireversing Techniques to Wintel Machine Code

Extreme care must be taken when applying antireversing techniques because some ultimately change the machine code or Java bytecode that will be executed on the target processor. In the end, if a program doesn’t work, measuring how efficient or difficult to reverse engineer it is becomes meaningless [18]. Some of the antireversing transformations performed on source code to make it more difficult to understand in both source and executable formats, can make the source code more challenging for a compiler to process because the program no longer looks like something a human would write. [18] states “any compiler is going to have at least some pathological programs which it will not compile correctly.” Compiler failures on so called “pathological” programs occur because compiler test cases are most often coded by people—not mechanically generated by a tool that knows how to try every fringe case and surface every bug. Keeping this in mind, one should no be surprised if some compilers have difficulty with obfuscated source code. Following the basic antireversing techniques introduced in section 6, we now investigate the technique Eliminating Symbolic Information as it applies to Wintel machine code.

7.1 Eliminating Symbolic Information in Wintel Machine Code

Eliminating Symbolic Information calls for the removal of any meaningful symbolic information in the machine code that is not important to the execution of the program, but serves to ease debugging or reuse of it by another program. For example, if a program relies on certain function or methods names (as a DLL does) the names of those methods or functions will appear in the .idata (import data) section of the Windows PE header. In production versions of a program, the machine code doesn’t directly contain any symbolic information from the original source code--such as method names, variable names, or line numbers; the executable file only contains the machine instructions that were produced by the compiler [9]. This lack of information about the connection between the machine instructions and the original source is unacceptable for purposes of debugging—this is why most modern compilers, like GCC, include an option to generate debugging information into the executable file that allow one to trace a failure occurring at a particular machine instruction back to a line in the original source code [9].

To show the various kinds of symbolic information that is inserted into machine code to enable debugging of an application, the GNU C++ compiler was directed to compile the program Calculator.cpp with debugging information but to generate assembly language instead of machine code. The source code for Calculator.cpp and the generated assembly language equivalent are given in Table 7.1. The GNU compiler stores debug information in the symbol tables (.stabs) section of the Windows PE header so that it will be loaded into memory as part of the program image. It should be clear
from the generated assembly in Table 7.1 that the debugging information inserted by 
GCC is by no means a replacement for the original source code of the program. A 
source-level debugger, like the GNU Project Debugger (GDB), must be able to locate 
the original source code file to make use of the debugging information embedded in the 
executable. Nevertheless, debugging information can give plenty of hints to a reverse 
engineer, such as the count and type of parameters one must pass to a given method. An 
obvious recommendation to make here, assuming there is an interest in protecting 
machine code from being reverse engineered, is to ensure that source code is not 
compiled for debugging when generating machine code for use by customers.

Table 7.1. Debugging information inserted into machine code.

<table>
<thead>
<tr>
<th>Calculator.cpp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: int main(int argc, char *argv[])</td>
</tr>
<tr>
<td>02: {</td>
</tr>
<tr>
<td>03: string input; int op1, op2; char fnc; long res;</td>
</tr>
<tr>
<td>04: cout &lt;&lt; &quot;Enter integer 1: &quot;;</td>
</tr>
<tr>
<td>05: getline(cin, input); op1 = atoi(input.c_str());</td>
</tr>
<tr>
<td>06: cout &lt;&lt; &quot;Enter integer 2: &quot;;</td>
</tr>
<tr>
<td>07: getline(cin, input); op2 = atoi(input.c_str());</td>
</tr>
<tr>
<td>08: cout &lt;&lt; &quot;Enter function [+</td>
</tr>
<tr>
<td>09: getline(cin, input); fnc = input.at(0);</td>
</tr>
<tr>
<td>10: switch (fnc)</td>
</tr>
<tr>
<td>11: {</td>
</tr>
<tr>
<td>12: case '+':</td>
</tr>
<tr>
<td>13: res = doAdd(op1, op2); break;</td>
</tr>
<tr>
<td>14: case '-':</td>
</tr>
<tr>
<td>15: res = doSub(op1, op2); break;</td>
</tr>
<tr>
<td>16: case '*':</td>
</tr>
<tr>
<td>17: res = doMul(op1, op2); break;</td>
</tr>
<tr>
<td>18: }</td>
</tr>
<tr>
<td>19: cout &lt;&lt; &quot;Result: &quot; &lt;&lt; res &lt;&lt; endl;</td>
</tr>
<tr>
<td>20: return 0;</td>
</tr>
<tr>
<td>21: }</td>
</tr>
<tr>
<td>22: long doAdd(int op1, int op2) { return op1 + op2; }</td>
</tr>
<tr>
<td>23: long doSub(int op1, int op2) { return op1 - op2; }</td>
</tr>
<tr>
<td>24: long doMul(int op1, int op2) { return op1 * op2; }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calculator.s (abbreviated assembly):</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: .file &quot;Calculator.cpp&quot;</td>
</tr>
<tr>
<td>02: .stabs &quot;C:/SRECD/MiscCPPSource/Calculator/&quot;,100,0,0,Ltext0</td>
</tr>
<tr>
<td>03: .stabs &quot;Calculator.cpp&quot;,100,0,0,Ltext0</td>
</tr>
<tr>
<td>04: .stabs &quot;main:F(0,3)&quot;,36,0,12, _main</td>
</tr>
<tr>
<td>05: .stabs &quot;argc:p(0,3)&quot;,160,0,12,8</td>
</tr>
<tr>
<td>06: .stabs &quot;argv:p(40,35)&quot;,160,0,12,12</td>
</tr>
<tr>
<td>06: __main:</td>
</tr>
<tr>
<td>07: .stabs &quot;Calculator.cpp&quot;,132,0,0,Ltext</td>
</tr>
<tr>
<td>08: call _Z5doAddii</td>
</tr>
</tbody>
</table>

22
The hunt for symbolic information doesn't end with information embedded by debuggers, it continues on to include the most prolific author of such helpful information—the programmer. Recall that in the animated tutorial on reversing Wintel machine code (see section 4) the key piece of information that led to the solution was the trial limitation message found in the .rdata (read-only) section of the executable. One can imagine that something as simple as having the Password Vault application load the trial limitation message from a file each time it’s needed and immediately clearing it from memory would have prevented the placement of a memory breakpoint on the trial message, which was an anchor for the entire tutorial. An alternative to moving the trial limitation message out of the executable would be to encrypt it so that a search of the dump would not turn up any hits; of course encrypted symbolic information would need to be decrypted before it is used. Encryption of symbolic information, as was discussed in relation to the Wintel animated tutorial, is an activity related to the obfuscation of a program, which we discuss next.

7.2 Basic Obfuscation of Wintel Machine Code

Obfuscating the Program calls for performing transformations to the source code and/or machine code that would render either extremely difficult to understand but functionally equivalent to the original. There are many kinds of transformations one can apply with varying levels of effectiveness, and as [5] states “an obfuscation transformation will typically have an associated cost (such as): larger code, slower execution time, or increased runtime memory consumption (by the machine code).”
Because of the high-level nature of intermediate languages like Java and .NET bytecode, there are free obfuscation tools that can perform fairly robust transformations on bytecode so that any attempt to decompile the program will still result in source code that compiles, but is near impossible to understand because of the obfuscation techniques that are applied. [19] states “Obfuscation (of Java bytecode) is possible for the same reasons that decompiling is possible: Java bytecode is standardized and well documented.”

Unfortunately, the situation is very different for machine code because it is not standardized; instruction sets, formats, and program image layouts vary depending on the target platform architecture. The side-effect of this is that tools to assist with obfuscating machine code are much more challenging to implement and expensive to acquire; no free tools were found at the time of this writing. One such commercial tool, EXECryptor (www.strongbit.com) is an industrial-strength machine code obfuscator that when applied to the machine code for the Password Vault application rendered it extremely difficult to understand. The transformations performed by EXECryptor caused such extreme differences in the machine code, including having compressed parts of it, that it was not possible to line up the differences between the original and obfuscated versions of the machine code to show evidence of the obfuscations. Therefore, to demonstrate machine code obfuscations in a way that is easy to follow, we'll perform obfuscations at the source code level and observe the differences in the assembly language generated by the GNU C++ compiler. The key idea here is that the obfuscated program has the same functionality as the original, but is more difficult to understand during live or static analysis attempts. There are no standards for code obfuscation, but it's relatively important to ensure that the obfuscations applied to a program are not easily undone because deobfuscation tools can be used to eliminate easily identified obfuscations [5].

Table 7.2 contains the source code and disassembly of VerifyPassword.cpp, a simple C++ program that contains an insecure password check that is no weaker than the implementation of the Password Vault trial limitation check. To find the relevant parts of .text and .rdata sections that are related to the password check, the now familiar technique of setting a breakpoint on a constant in the .rdata section was used.

**Table 7.2.** Listing of VerifyPassword.cpp and disassembly of VerifyPassword.exe.

VerifyPassword.cpp:

```cpp
01: int main(int argc, char *argv[])  
02: {  
03:   const char *password = "jup!ter";  
04:   string specified;  
05:   cout << "Enter password: ";  
06:   getline(cin, specified);  
07:   if (specified.compare(password) == 0)  
08:   {  
09:     cout << "[OK] Access granted." << endl;  
10:   } else
```


Using the simple program `VerifyPassword.cpp`, we now investigate applying obfuscations to make machine code more difficult to reverse engineer. The first obfuscation that will be applied is a data transformation technique which [5] calls “Modifying Variable Encoding”. Essentially this technique prescribes that all meaningful and sensitive constants in a program be stored or represented in an alternate encoding, such as ciphertext. For numerics, one can imagine storing or working with a function of a number instead of the number itself; for example, instead of testing for \( \alpha < 10 \), we can obscure the test by checking if \( 1.2^\alpha < 1.2^{10} \) instead. To make string constants unreadable in a dump of the .rdata section we can employ a simple substitution cipher whose decryption function would become part of the machine code. A simple substitution cipher is an encryption algorithm where each character in the original string is replaced by another using a one-to-one mapping [20]. Substitution ciphers are easily broken because the algorithm is the secret [21], so while we will use one for ease of demonstration, stronger encryption algorithms should be used in real-world scenarios.

Table 7.3 contains the definition of a simple substitution cipher that shifts each character 13 positions to the right in the local 8-bit ASCII or EBCDIC character set. Ciphertext is generated or read in printable hexadecimal to allow all members of the character set, including control characters, to be used in the mappings. Note: unlike
ROT13 [22], this cipher is not its own inverse—meaning that shifting each character an additional 13 positions to the right will not perform decryption.

**Table 7.3.** Simple substitution cipher used to protect string constants.

<table>
<thead>
<tr>
<th>SubstitutionCipher.h:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: class SubstitutionCipher</td>
</tr>
<tr>
<td>02: {</td>
</tr>
<tr>
<td>03: public:</td>
</tr>
<tr>
<td>04: SubstitutionCipher();</td>
</tr>
<tr>
<td>05: string encryptToHex(string plainText);</td>
</tr>
<tr>
<td>06: string decryptFromHex(string cipherText);</td>
</tr>
<tr>
<td>07: private:</td>
</tr>
<tr>
<td>08: unsigned char encryptTable[256];</td>
</tr>
<tr>
<td>09: unsigned char decryptTable[256];</td>
</tr>
<tr>
<td>10: char hexByte[2];</td>
</tr>
<tr>
<td>11: );</td>
</tr>
</tbody>
</table>

**Full source code:**

http://reversingproject.info/repository.php?fileID=7_2_1

Using the substitution cipher given in Table 7.3, we replace each string constant in `VerifyPassword.cpp` with its equivalent ciphertext. Even strings with format modifiers such as “%s” and “%d” can be encrypted as these inserts are not interpreted by methods such as `printf` and `sprintf` until execution time. Table 7.4 contains the source code and disassembly for `VerifyPasswordObfuscated.exe`, where each string constant in the program is stored as ciphertext; when the program needs to display a message, the ciphertext is passed to the bundled decryption routine. The transformation we've manually applied removes the helpful information the string constants provided when they were stored in the clear. Given that modern languages have well-documented grammars, it should be possible to develop a tool that automatically extracts and replaces all string constants with ciphertext that is wrapped by a call to the decryption routine.

**Table 7.4.** `VerifyPasswordObfuscated.cpp` and disassembly of `VerifyPasswordObfuscated.exe`.

<table>
<thead>
<tr>
<th>VerifyPasswordObfuscated.cpp:</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: #include &quot;substitutioncipher.h&quot;</td>
</tr>
<tr>
<td>02: using namespace std;</td>
</tr>
<tr>
<td>03: static const char *password = &quot;77827D2E81727F&quot;;</td>
</tr>
<tr>
<td>04: static const char *enter_password = &quot;527B81727F2D7D6E8080847C7F71472D&quot;;</td>
</tr>
<tr>
<td>05: static const char *password_ok = &quot;685C586A2D4E70707280802D747F6E7B8172713B&quot;;</td>
</tr>
<tr>
<td>06: static const char *password_bad = &quot;68527F77C7F6A2D4E7070728082D71727B7672713B&quot;;</td>
</tr>
<tr>
<td>07: int main(int argc, char *argv[])</td>
</tr>
</tbody>
</table>

26
Once all constants have been stored in an alternate encoding, the next step one could take to further protect the `VerifyPassword.cpp` program would be to obfuscate the condition in the code that tests for the correct password. Applying transformations to disguise key logic in a program is an activity related to the antireversing technique `Obfuscating the Program`. For purposes of demonstration we'll implement some obfuscations to the trial limitation check in the C++ version of the `Password Vault` application, which was introduced in section 4, but first we discuss an additional application of the technique `Obfuscating the Program` that helps protect intellectual property when proprietary software is shipped as source code.

### 7.3 Protecting Source Code Through Obfuscation

When delivering a software application to clients, there may exist a requirement to ship the source code so that the application binary can be created on the clients' computer using shop-standard build and audit procedures. If the source code contains intellectual property that is worth protecting, one can perform transformations to the source code which make it difficult to read, but have no impact on the machine code that would ultimately be generated when the program is compiled. To demonstrate source code obfuscation, COBF [23], a free C/C++ source code obfuscator was configured and given `VerifyPassword.cpp` as input; the results of which are displayed in Table 7.5.
Table 7.5. COBF obfuscation results for VerifyPassword.cpp.

COBF invocation:

01: C:\cobf_1.06\src\win32\release\cobf.exe
02: @C:\cobf_1.06\src\setup_cpp_tokens.inv -o cobfoutput -b -p C:
03: \cobf_1.06\etc\pp_eng_msvc.Bat VerifyPassword.cpp

COBF obfuscated source for VerifyPassword.cpp:

01: #include"cobf.h"
02: ls lp lk;lf lo(lf ln,ld*lj[]){ll ld*lc="\x6a\x75\x70\21\x74
03: \x65\x72";lh 1a;lb="\x45\x6e\74\x65\72\x20\x70\x61\x73\x77"
04: \x6f\x72\x64"\x3a\20";li(lq,la);lm(la.lg(lc)==0){lb="\x5b
05: \x4f\x4b\5d\x20\x41" "%x63\x65\x73\x77\x3a\20\x67\x72\x61\x6e
06: \x74\x65\x64\x2e"<<le;}lr{lb="\x5b\x45\x72\x72\x5d\x20\x41\x63"
07: \x20\x6e"\x69\x65\x64" "%x65\x6e\x69\x65
08: \x64\x2e"<<le;}}

COBF generated header (cobf.h):

01: #define ls using
02: #define lp namespace
03: #define lk std
04: #define lf int
05: #define lo main
06: #define ld char
07: #define ll const
08: #define lh string
09: #define lb cout
10: #define li getline
11: #define lq cin
12: #define lm if
13: #define lg compare
14: #define le endl
15: #define lr else

COBF replaces all user-defined method and variables in the immediate source file with meaningless identifiers. In addition, COBF replaces standard language keywords and library calls with meaningless identifiers, however these replacements must be undone before compilation; for example, the keyword “if” cannot be left as “lm”. Therefore, COBF generates the cobf.h header file which includes the necessary substitutions to make the obfuscated source compilable. Through this process, all user-defined method and variable names within the immediate file are lost, rendering the source code difficult to understand, even if one performs the substitutions prescribed in cobf.h. Since COBF generates obfuscated source as a continuous line, any formatting in the source code that served to make it more readable is lost. While the original formatting cannot be recovered, a code formatter such as Artistic Style [24] can be used to format the code using ANSI formatting schemes so that methods and control structures can again be identified via visual inspection. Source code obfuscation is a fairly weak form of intellectual property protection, but it does serve a purpose in real-world scenarios where a given application needs to be built on the end-user’s target computer—instead of being pre-built and delivered on installation media.
7.4 Advanced Obfuscation of Machine Code

One of the features of an interactive debugger-disassembler like OllyDbg that is very helpful to a reverse engineer is the ability to trace the machine instructions that are executed when a particular operation or function of a program is tried. In the Password Vault application, introduced in section 4, a reverse engineer could pause the program's execution in OllyDbg right before specifying the option to create a new password record. To see which instructions are executed when the trial limitation message is displayed, the reverser can choose to record a trace of all the instructions that are executed when execution is resumed. To make it difficult for a reverse engineer to understand the logic of a program through tracing or stepping through instructions, we can employ control flow obfuscations, which introduce confusing, randomized, benign logic that serves to make live and static analysis (debugging and tracing) difficult. The often randomized and recursive nature of effective control flow obfuscations can make traces more difficult to understand and interactive debugging sessions less helpful: randomization makes the execution of the program appear different each time it's run, while recursion makes stepping through code more difficult because of deeply nested procedure calls.

In [5], three types of control flow transformations are introduced: computation, aggregation, and ordering. Computation transformations reduce the readability of machine code and, in the case of opaque predicates, can make it difficult for a decompiler to generate equivalent high-level language source code. Aggregation transformations attempt to remove the high-level structure of a program as it's translated to machine code; this serves to defeat attempts to reconstruct, either mentally or programmatically, the high-level organization of the code. Ordering transformations randomize the order of operations in a program to make it more difficult to follow the logic of a program during live or static analysis (debugging or tracing). To provide a concrete example of how control flow obfuscations can be applied to protect a non-trivial program, we'll apply both a computation and ordering control flow obfuscation to the trial limitation check in the Password Vault application and analyze their potential effectiveness by gathering some statistics on the execution of the obfuscated trial limitation check.

7.5 Wintel Machine Code Antireversing Exercise

Apply the antireversing techniques Eliminating Symbolic Information and Obfuscating the Program, both introduced in sections 6 and 7, to the C/C++ source code of the Password Vault application with the goal of making it more difficult to disable the trial limitation. Rebuild the executable binary for the Password Vault application from the modified sources using the GNU compiler collection for Windows. Show that the Wintel machine code reversing solution shown in the animated tutorial in section 4.4 can no longer be carried out as demonstrated.

7.6 Solution to the Wintel Antireversing Exercise

The solution to the Wintel machine code antireversing exercise is given through
comparisons of the original and obfuscated source code of the Password Vault application. As each antireversing transformation is applied to the source code, important differences and additions are explained through a series of generated diff reports and memory dumps. Once the antireversing transformations have been applied, we cover the impact they have on the machine code and how reversing the Password Vault application becomes more difficult when these obfuscations make it difficult to find a good starting point and hinder live and static analysis. The obfuscated source code for the Password Vault application is located in the `obfuscated_source` directory of the archive located at http://reversingproject.info/repository.php?fileID=4_1_2.

### 7.6.1 Encryption of String Literals

To eliminate the obvious starting point of setting an access breakpoint on the trial message, all of the messages issued by the application are stored as encrypted hexadecimal literals that are decrypted each time they are used—keeping the decrypted versions out of memory as much as possible. Table 7.6 gives an example of the needed code changes to `PasswordVaultConsoleUtil.cpp`.

#### Table 7.6. Encrypted strings are decrypted each time they are displayed.

<table>
<thead>
<tr>
<th>Line</th>
<th>Original Code</th>
<th>Obfuscated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>133</td>
<td><code>case __createPasswordRecord: return  &quot;Create a Password Record&quot;;</code></td>
<td><code>case __createPasswordRecord: DecryptMessageText(&quot;507F726E81722D6E2D56E8080847C7F712D5F72707C7F71&quot;, _textBuffer);</code></td>
</tr>
<tr>
<td>186</td>
<td><code>case __recordLimitReached: return  &quot;Thank you for trying Password Vault! You have reached the maximum number of records allowed in this trial version.&quot;;</code></td>
<td><code>case __recordLimitReached: DecryptMessageText(&quot;61756E7B782D867C822D737C7F2D817F86767B742D56E8080847C7F712D636E827891822D667C822D756E83722D7F726E707572712D817572D7A685767A827A2D7BB27A6F727F2D7C732D7F72707C7F71802D6E79797C8472712D767B2D817576802D817F766792D83727F80767C7B3B&quot;, _textBuffer);</code></td>
</tr>
</tbody>
</table>

The net effect of encrypting the literals is shown in Figure 7.1 where a dump of the `.rdata` section of the Password Vault program image no longer yields the clues it once did.

30
Since the literals are no longer readable, one cannot simply locate and set a breakpoint on the trial limitation message—as was done in the solution to the Wintel machine code reversing exercise—causing a reverser to choose an alternate strategy. Note that more than just the trial limitation message would need to be encrypted otherwise it would look quite suspicious in a memory dump alongside other non-encrypted strings!

![Figure 7.1. Result of obfuscating all string literals in the program.](image)
7.6.2 Obfuscating the Numeric Representation of the Record Limit

Having obfuscated the string literals in the program image, we'll assume that a reverse engineer will need to select the alternate strategy of pausing the program's execution immediately before specifying the input that causes the trial limitation message to be displayed. Using this strategy, a reverser can either capture a trace of all the machine instructions that are executed when the trial limitation message is displayed, or debug the application—stepping through each machine instruction until a sequence that seems responsible for enforcing the trial limitation is reached. Recall that in the solution to the Wintel machine code reversing exercise, an obvious instruction sequence that tested a memory location for a limit of five password records was found. By using an alternate but equivalent representation of the record limit we can make the record limit test a bit less obvious. The technique we employ here is to use a function of the record limit instead of the actual value; for example, instead of testing for \( \alpha \leq 5 \), where \( \alpha \) is the record limit, we obscure the limit by testing if \( 2^{\alpha} \leq 2^{5} \). Table 7.7 gives an example of the needed code changes to *PasswordVault.cpp*.

**Table 7.7.** Encrypted strings are decrypted each time they are displayed.

```c
176  void PasswordVault::doCreateNewRecord()
178 #ifdef TRIALVERSION
180     // Add limit on record count for reversing exercise
181     if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT)
181     ==> 181 if ((pow(2.0, (double)passwordStore.getRecords().size()) >=
180                 pow(2.0, 5.0)))
```

The effects of the source code changes in Table 7.7 on the machine code are shown in Figure 7.2. A function of the record limit is referenced during execution instead of the limit itself. This type of obfuscation is as strong as the function used to obscure the actual condition is to unravel. Keep in mind that a reverse engineer will not have the non-obfuscated machine code for reference, so even a very weak function, like the one used in this solution, may be effective at wasting some of a reverser's time. The numeric function used here is very simple; more complex functions can be devised that would further decrease the readability of the machine code.

7.6.3 Control Flow Obfuscation for the Record Limit Check

We introduce some non-essential, recursive, and randomized logic to the password limit check in *PasswordVault.cpp* to make it more difficult for a reverser to perform static or live analysis. A design for obfuscated control flow logic which ultimately implements the trial limitation check is given in Figure 7.3. Since no standards exist for control flow obfuscation, this algorithm was designed by the author using the cyclomatic complexity metric defined by McCabe [24] as a general guideline for creating a highly-complex control flow graph for the trial limitation check.
Figure 7.2. Record limit operands are represented as exponents with a base of 2.

The record limit check is abstracted out into the method `isRecordLimitReached` which returns whether or not the record limit is reached after having invoked the method `isRecordLimitReached_0`. The method `isRecordLimitReached_0` invokes itself recursively a random number of times, growing the call stack by a minimum of 16 and a maximum of 64 frames. Each invocation of `isRecordLimitReached_0` tests whether the record limit has been reached, locally storing the result, before randomly invoking one of the methods `isRecordLimitReached_1`, `isRecordLimitReached_2`, or `isRecordLimitReached_3`. When the call stack is unraveled, `isRecordLimitReached_0` finally returns whether or not the record limit is reached in the method `isRecordLimitReached`. Table 7.8 shows the required code changes to implement the control flow obfuscation. Note that a sum of random numbers returned from methods `isRecordLimitReached_1`, `isRecordLimitReached_2`, and `isRecordLimitReached_3`
stored in `randCallSum`, a private attribute of the class; this is to protect against a compiler optimizer discarding the calls because they would otherwise have no effect on the state of any variables in the program.

![Diagram of control flow logic]

**Figure 7.3.** Obfuscated control flow logic for testing the password record limit.

**Table 7.8.** PasswordVault.cpp: implementation of the control flow obfuscation in Figure 7.4.

```cpp
if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT) => if (isRecordLimitReached())

01: bool PasswordVault::isRecordLimitReached()
02: {
03:     srand(time(NULL));
```
controlFlowAltRemain = max(4, abs(rand()) % 64);
return isRecordLimitReached_0();

bool PasswordVault::isRecordLimitReached_0()
{
while (controlFlowAltRemain > 0)
{
  controlFlowAltRemain--;
isRecordLimitReached_0();
}

bool reached = (pow(2.0, (double)passwordStore.getRecords().size()) >= pow(2.0, 5.0));

randCallSum = 0;

switch (abs(rand()) % 3)
{
case 0:
  randCallSum += isRecordLimitReached_1();
  break;
case 1:
  randCallSum += isRecordLimitReached_2();
  break;
case 2:
  randCallSum += isRecordLimitReached_3();
  break;
}

return reached;

unsigned int PasswordVault::isRecordLimitReached_1()
{
return abs(rand());
}

unsigned int PasswordVault::isRecordLimitReached_2()
{
return abs(rand());
}

unsigned int PasswordVault::isRecordLimitReached_3()
{
return abs(rand());
}
7.6.4 Analysis of the Control Flow Obfuscation Using Run Traces

The goal of this analysis is to demonstrate that even though the Password Vault application is given identical input and delivers identical output on subsequent runs, OllyDbg run traces, which contain the executed sequence of assembly instructions, will be significantly different from each other—making it difficult for a reverser to understand the trial limitation check through live or static analysis of the disassembly. Live analysis is hampered more by randomization than static analysis is because the control flow of the trial limitation check is randomized each time it is run; one can imagine the confusion that would arise if breakpoints are not always triggered, or triggered in an unpredictable order.

OllyDbg run traces are captured using the run trace view once the execution of a program has been paused at the desired starting point. To have the trace logged to a file in addition to the view, select “log to file” on the context menu of the run trace view. Begin the trace by selecting “Trace into” on the “Debug” menu; the program will execute, but much more slowly than normal since each instruction must be inspected and added to the run trace view and optional log file. An OllyDbg trace will include all the instructions executed by the program and its operating system dependencies; fortunately the trace is columnar with each instruction qualified by the name of the module that executed it, so it is possible to post process the trace and extract only those instructions executed by a particular module of interest. For example, in the case of the Password Vault traces which we will analyze in this section, the Sed (stream-editor) utility was used to filter the run traces—leaving only instructions executed by the “Password” module.

To analyze the effectiveness of the ordering (control flow) obfuscation, statistics on the differences between three different run traces were gathered using a modification of Levenshtein Distance (LD), a generalization of Hamming Distance, to compute the edit-distance—the number of assembly instruction insertions, deletions, or substitutions needed to transform one trace into the other; we’ve modified LD to consider each instruction instead of each character in the run traces. Figure 7.4 illustrates the significant differences that exist between the traces at the point of the obfuscated trial limitation check. The randomized control-flow obfuscation causes significant differences in subsequent executions of the trial limitation check—hopefully creating enough of a deterrent for a reverse engineer by hampering live and static analysis efforts. Table 7.9 contains the statistical data that was gathered for the analysis.

A C++ implementation of Levenshtein Distance, written for this solution, can be downloaded from http://reversingproject.info/repository.php?fileID=7_6_1. Note that computing the edit-distance between two large files of any type can take many hours on a modern PC. For reference, the average size of three traces analyzed in this section is 10MB, and to compute the edit-distance between two of them required an average of ~20 hours of CPU time on an Intel Pentium 1.6GHz Dual-core processor. The LD implementation employed in this analysis uses a dynamic-programming approach that requires O(m) space; note that some reference implementations of LD require O(mn).
space since they use a \((m + 1) \times (n + 1)\) matrix which is impractical for large files [25]. The \(~20\) hour execution time for the LD implementation is mainly because the dynamic programming algorithm is quite naïve; perhaps an approximation algorithm would perform significantly better.

**Analysis of Run Trace Levenshtein Edit Distances**

![Graph showing edit distances between three run traces of the trial limitation check.](image)

**Table 7.9.** Statistical data gathered for randomized control-flow obfuscation.

<table>
<thead>
<tr>
<th>Trace Comparison</th>
<th>Levenshtein Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace #1 (</td>
<td></td>
</tr>
</tbody>
</table><p>ightarrow) Trace #2 | 101414 |
| Trace #2 (ightarrow) Trace #3 | 67590 |
| Trace #1 (ightarrow) Trace #3 | 168892 |</p>

<table>
<thead>
<tr>
<th>Trace Comparison</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace #1 (</td>
<td></td>
</tr>
</tbody>
</table><p>ightarrow) Trace #2 | 7932.32 |
| Trace #2 (ightarrow) Trace #3 | 31849.5 |</p>

**Figure 7.4.** Edit-distances between three run traces of the trial limitation check.
8 Applying Antireversing Techniques to Java Bytecode

It was demonstrated in the Java reversing and patching exercise of section 5.2 that decompilation of Java bytecode to Java source code is possible with quite good results. While it is most often the case that we cannot recover the original Java source code from the bytecode, the results will be functionally equivalent. When new features are added to the Java language they won't always introduce new bytecode instructions; for example, support for generics is implemented by carrying additional information in the constants pool of the bytecode that describes the type of object a collection should contain; this information can then be used at execution time by the JVM to validate the type of each object in the collection. The strategy of having newer Java language constructs result in compatible bytecode with optionally-utilized metadata provides the benefit of allowing legacy Java bytecode to run on newer JVMs, however if a decompiler doesn't know to look for the metadata, some information is lost; for example, the fact that a program used generics would not be recovered and all collections would be of type `Object` (with cast statements of course).

Recall that in section 4.1 the Boomerang decompiler failed to decompile the machine code for a simple C/C++ “Hello World” program, however in section 5.1, the Jad decompiler produced correct Java source code for a slightly larger program. Given these results, one does need to be concerned with with protecting Java bytecode from decompilation if there is significant intellectual property in the program. The techniques used to protect machine code in the antireversing exercise solution, detailed in section 7.6, can also be applied to Java source code to produce bytecode that is obfuscated. Since Java bytecode is standardized and well-documented there are many free Java obfuscation tools available on the Internet such as SandMark [27], ProGuard [29], and RetroGuard [28] which perform transformations directly on the Java bytecode instead of on the Java source code itself. Obfuscating bytecode is inherently easier than obfuscating source code because bytecode has a significantly more strict and organized representation than source code—making it much more easy to parse. For example, instead of parsing through Java source code looking for string constants to encrypt (protect), one can easily look in the constant pool section of the bytecode. The constant pool section of a Java Class File, unlike the .rdata section of Wintel machine code, contains a well-documented table data structure that makes available the name and length of each constant; on the other hand, the .rdata section of Wintel machine code simply contains all the constants in the program in a contiguous, unstructured bytestream. The variable names, method names, and string literals, in the constant pool section of Java bytecode provide a wealth of information to a reverse engineer regarding the structure and operation of the bytecode and hence should be obfuscated to protect the software. Therefore, we now look at
applying the technique *Eliminating Symbolic Information* in the context of Java bytecode.

### 8.1 Eliminating Symbolic Information in Java Bytecode

Variable, class, and method names, are all left intact when compiling Java source code to Java bytecode. This is a stark difference from machine code where variable and method names are not preserved. Sun Microsystem's Java compiler, javac, provides an option to leave out debugging information in Java bytecode: specifying javac -g:none will exclude information on line numbers, the source file name, and local variables. This option offers little to no help in fending off a reverse engineer since none of the variable names, methods names, or string literals are obfuscated. According to the documentation for Zelix Klassmaster [26], a Java bytecode obfuscation tool, a high-level of protection can be achieved for Java bytecode by applying three transformations: (1) Name Obfuscation, (2) String Encryption, and (3) Flow Obfuscation. Unfortunately, at the time of this writing, no free-of-charge software tool was found on the Internet that can perform all three of these transformations to Java bytecode. A couple of tools, namely ProGuard [29] and RetroGuard [28] are capable of applying transformation (1), and SandMark [27], a Java bytecode watermarking and obfuscation research tool, is capable of applying transformation (2), although not easily. Experimentation with SandMark V3.4 was not promising since its “String Encoder” obfuscation function only worked on a trivial Java program; it failed when given more substantial input such as some of the classes that implement Java version of the Password Vault application. It's clear from a survey of existing Java bytecode obfuscators that a full-function, robust, open-source bytecode obfuscator is sorely needed. Zelix Klassmaster, a commercial product capable of all the three transformations mentioned above, is said to be the best overall choice of Java bytecode obfuscator in [19]. A 30-day evaluation version of Zelix Klassmaster can be downloaded from the company's web site.

Of course one can always make small-scale modifications to Java bytecode with a bytecode editor such as CafeBabe [30]. Incidentally, CafeBabe gets its catchy name from the fact that the hexadecimal value 0xCAFEBABE comprises the first four bytes of every Java class file; this value is known as the “magic number” which identifies every valid Java class file. To demonstrate applying transformations to Java bytecode, we'll target the bytecode for program *CheckLimitation.java* whose source code is given in Table 8.1; for this demonstration, assume that a reverse engineer is interested in eliminating the limit on the number of passwords and that we are interested in protecting the software.

<table>
<thead>
<tr>
<th>Table 8.1. Unobfuscated source listing of CheckLimitation.java.</th>
</tr>
</thead>
<tbody>
<tr>
<td>01: public class CheckLimitation {</td>
</tr>
<tr>
<td>02:</td>
</tr>
<tr>
<td>03: private static int MAX_PASSWORDS = 5;</td>
</tr>
<tr>
<td>04: private ArrayList&lt;String&gt; passwords;</td>
</tr>
<tr>
<td>05:</td>
</tr>
<tr>
<td>06: public CheckLimitation()</td>
</tr>
</tbody>
</table>
We begin obfuscating CheckLimitation.java by applying transformation (1) Name Obfuscation: rename all variables and methods in the bytecode so they no longer provide hints to a reverser when the bytecode is decompiled or edited. Using ProGuard, we obfuscate the bytecode and then decompile it using Jad to observe the effectiveness of the obfuscation; the result of decompiling the obfuscated bytecode using Jad is given Table 8.2. As expected, all user-defined variable and method names have been changed to meaningless ones; of course the names of Java standard library methods must be left as-is. ProGuard seems to use a different obfuscation scheme for local variables within a method; it's not clear why the variable “loop” in the main method has been changed to “flag” since it's still a very descriptive name.

Table 8.2. Jad decompilation of ProGuard obfuscated bytecode.

```
01: public class CheckLimitation {
02:    private static int a = 5;
03:    private ArrayList b;
04:    public CheckLimitation()
05:    {
06:    }
```
\[
\begin{align*}
07: & \quad \{ \\
08: & \quad b = \text{new ArrayList();} \\
09: & \quad \} \\
10: & \\
11: & \quad \text{public boolean } a(\text{String } s) \\
12: & \quad \{ \\
13: & \quad \quad \text{if (} b.\text{size()} \geq a \) \\
14: & \quad \quad \{ \\
15: & \quad \quad \quad \text{System.out.println(} \text{"[Error] The maximum number of passwords has been exceeded!"}) \\
16: & \quad \quad \quad \text{return false;} \\
17: & \quad \quad \} \quad \text{else} \\
18: & \quad \{ \\
19: & \quad \quad b.\text{add}(s); \\
20: & \quad \quad \text{System.out.println(} (\text{new StringBuilder()}).append(\text{"[Info] password("").append(s).append(\text{") added successfully."}).toString()); \\
21: & \quad \quad \text{return true;} \\
22: & \quad \} \\
23: & \} \\
24: & \\
25: & \quad \text{public static void main(\text{String } args[])} \\
26: & \quad \{ \\
27: & \quad \quad \text{CheckLimitation checklimitation} = \text{new CheckLimitation();} \\
28: & \quad \quad \text{boolean } flag = \text{true;} \\
29: & \quad \quad \text{for(int } i = 0; i < \text{args.length } \&\& \text{ flag; } i++) \\
30: & \quad \quad \quad \text{if(!checklimitation.a(args[i])) } flag = \text{false;} \\
31: & \quad \} \\
32: & \\
33: & \}
\end{align*}
\]

Next we further obfuscate the bytecode by applying transformation (2) String Encryption, and we do so by employing the “String Encoder” obfuscation in SandMark to protect the string literals in the program from being understood by a reverser. The “String Encoder” function in SandMark implements an encryption strategy for literals in the bytecode that is similar to the one which was demonstrated at the source code level in the Wintel machine code anti-reversing background section: each string literal is stored in a weakly encrypted form and decrypted on-demand by a bundled decryption function. Table 8.3 contains the Jad decompilation result for the \textit{CheckLimitation.java} bytecode that was first obfuscated using ProGuard and subsequently obfuscated using the “String Encoder” functionality in SandMark.

\begin{table}[h]
\centering
\begin{tabular}{|l|}
\hline
01: \text{public class CheckLimitation} \{ \\
02: \\
03: \quad \text{private static int } a = 5; \\
04: \quad \text{private ArrayList } b; \\
05: \} \\
\hline
\end{tabular}
\caption{Jad decompilation of SandMark (and ProGuard) obfuscated bytecode.}
\end{table}
public CheckLimitation()
{
    b = new ArrayList();
}

public boolean a(String arg0)
{
    if(b.size() >= a)
    {
        System.out.println(Obfuscator.DecodeString("\253\315\253\315\uFF9E\u2A31\u5D75\u2AB1\u3884\u91E0\u533C\u5654\uDF6E\uA919\uB6DE\u0CD9\u6769\1F26\u3581\uBD9D\uADE1"));
        return false;
    } else
    {
        b.add(arg0);
        System.out.println((new StringBuilder()).append(Obfuscator.DecodeString("\253\315\253\315\uFF9E\u2A31\u5D75\u2AB1\u3884\u91E0\u533C\u5654\uDF6E\uA919\uB6DE\u0CD9\u6769\1F26\u3581\uBD9D\uADE1")).append(arg0).append(Obfuscator.DecodeString("\253\315\253\315\uFFEC\u2A58\5D7A\u5378\56D7C\uA91F\u6769\1F27\u3596\uBD9D\uA919\uB6DE\u0CD9\u6769\1F26\u3581\uBD9D\uADE1")));
        return true;
    }
}

public static void main(String arg0[])
{
    CheckLimitation checklimitation = new CheckLimitation();
    boolean flag = true;
    for(int i = 0; i < arg0.length && flag; i++)
    {
        if(!checklimitation.a(arg0[i])) flag = false;
    }
}

We can see that each string literal is decrypted using the Obfuscator class which was generated by SandMark. Because Obfuscator is a public class, it must be generated into a separate file named Obfuscator.class—making it very straight-forward for a reverser to isolate, decompile, and learn the encryption algorithm. The danger of giving away the code for the string decryption algorithm is that it could then be used to programatically update the constants pool section of the bytecode to contain the plaintext versions of each string literal, essentially undoing the obfuscation. Ideally, we would like to prevent a reverser from being able to successfully decompile the obfuscated bytecode; this can be accomplished through control flow obfuscations which we explore next.
8.2 Preventing Decompilation of Java Bytecode

One of the most popular, and fragile, techniques for preventing decompilation involves the use of opaque predicates which introduce false ambiguities into the control flow of a program—tricking a decompiler into traversing garbage bytes that are masquerading as the logic contained in an else clause. Opaque predicates are false branches, branches that appear to be conditional but are really not [5]. For example, the conditions “if ( 1 == 1 )” and “if ( 1 == 2 )” implement opaque predicates because the first always evaluates to true, and the second always to false. The essential element in preventing decompilation with opaque predicates is to insert invalid instructions in the else branch of an always-true predicate (or the if-body of an always false predicate). Since the invalid instructions will never be reached during normal operation of the program there is no impact on the program's operation. The obfuscation only interferes with decompilation, where a naïve decompiler will evaluate both “possibilities” of the opaque predicate and fail on attempting to decompile the invalid, unreachable instructions. Figure 8.1 illustrates how opaque predicates would be used to protect bytecode from decompilation. Unfortunately, this technique, often used in protecting machine code from disassembly, cannot be used with Java bytecode because of the presence of the Java Bytecode Verifier in the JVM. Before executing bytecode the JVM performs the following checks using single-pass static analysis to ensure that the bytecode has not been tampered with; to understand why this is beneficial, imagine bytecode being executed as it's received over a network connection. [31] documents the following checks made by the Java Bytecode Verifier:

- **Type correctness**: arguments of an instruction, whether on the stack or in registers, should always be of the type expected by the instruction.
- **No stack overflow or underflow**: instructions which remove items from the stack should never do so when the stack is empty (or does not contain at least the number of arguments that the instruction will pop off the stack). Likewise, instructions should not attempt to put items on top of the stack when the stack is full (as calculated and declared for each method by the compiler).
- **Register initialization**: Within a single method any use of a register must come after the initialization of that register (within the method). That is, there should be at least one store operation to that register before a load operation on that register.
- **Object initialization**: Creation of object instances must always be followed by a call to one of the possible initialization methods for that object (these are the constructors) before it can be used.
- **Access control**: Method calls, field accesses, and class references must always adhere to the Java visibility policies for that method, field, or reference. These policies are encoded in the modifiers (private, protected, public, etc.).
Figure 8.1. Usage of opaque predicates to prevent decompilation.

Based on the high-level of bytecode integrity expected by the JVM, introducing garbage or illegal instructions into bytecode is not feasible. However, this technique does remain viable for machine code, though there is some evidence that good disassemblers, such as IDA Pro, do check for rudimentary opaque predicates [5]. The authors of SandMark claim that the sole presence of opaque predicates in Java bytecode, without garbage bytes of course, can make decompilation more difficult. Therefore, SandMark implements several different algorithms for sprinkling opaque predicates throughout bytecode. For example, SandMark includes an experimental “irreducibility” obfuscation function which is briefly documented as “insert jumps into a method via opaque predicates so that the control flow graph is irreducible. This inhibits decompilation.” Unfortunately this was not the case with the program DateTime.java shown in Table 8.4 as Jad was still able to decompile DateTime.class without any problems despite the changes made by SandMark’s “irreducibility” obfuscation. The bytes of the unobfuscated and obfuscated class files were compared to verify that SandMark did make significant changes; perhaps SandMark does work for special cases, so more investigation is likely warranted. In any event, opaque predicates seem to be far more effective when inserted into machine code because of the absence of any type of verifier that validates all machine instructions in a native binary before allowing it to execute.

Table 8.4. Listing of DateTime.java

Listing of DateTime.java (abbreviated):
01: public static void main(String arguments[])
02: {
03:   new DisplayDateTime().doDisplayDateTime();

44
public void doDisplayDateTime()
{
    Date date = new Date();
    System.out.println(String.format(DATE_TIME_MASK,
        date.toString()));
}

SandMark’s approach of using control flow obfuscations that leverage opaque predicates in an attempt to confuse a decompiler is not unique because Zelix Klassmaster, a commercial product, implements this approach as well. When Zelix Klassmaster V5.2.3a was given DateTime.class as input with both “aggressive” control flow and “String Encryption” selected, some interesting results were observed in the corresponding Jad decompilation. Table 8.5 lists the Jad decompilation of Zelix’s attempt at obfuscating DateTime.class. Zelix performed the same kind of name obfuscation seen with ProGuard, except it went a little too far and renamed the main method; this was corrected by manually adding an exception for methods named “main” in the tool. The results of the decompilation show that Zelix’s control flow obfuscation and use of opaque predicates is somewhat effective for this particular example because even though Jad was able to decompile most of the logic in DateTime.class, Zelix’s obfuscation caused Jad to lose the value of the constant DATE_TIME_MASK when using it on line 12, and generate a large block of static, invalid code starting at line 22. In the next two sections (8.3 and 8.4), a Java antireversing exercise with a complete animated solution is provided. In the solution, decompilation of Java bytecode is prevented through the use of a class encryption obfuscation implemented by SandMark. Issues regarding the use of this obfuscation technique are discussed in the animated solution.

Table 8.5. Jad decompilation of DateTime.class obfuscated by Zelix Klassmaster.

Listing of Jad decompilation of DateTime.class (abbreviated):

```java
public class a
{
    public static void main(String as[])
    {
        (new a()).a();
    }
    public void a()
    {
        boolean flag = c;
        Date date = new Date();
        System.out.println(String.format(a, new Object[] {
            date.toString()});
        if(flag)
```
15:       b = !b;
16:   }
17:
18:   private static final String a;
19:   public static boolean b;
20:   public static boolean c;
21:
22:   static
23:   {
24:     "?X@MA%O\005@@wY\001ZQw\\016J\024#T\rK\024>N@\013Gy";
25:     -1;
26:     goto _L1
27: _L5:
28:     a;
29:     break MISSING_BLOCK_LABEL_116;
30: _L1:
31:     JVM INSTR swap ;
32:     toCharArray();
33:     JVM INSTR dup ;

8.3 A Java Bytecode Code Antireversing Exercise

Use Java bytecode antireversing tools such as ProGuard, SandMark, and CafeBabe on the Java version of the Password Vault application to apply the antireversing techniques Eliminating Symbolic Information and Obfuscating the Program with the goal of making it more difficult to disable the trial limitation. Instead of attempting to implement a custom control flow obfuscation to inhibit static and dynamic analysis as was done in the solution to the machine code antireversing exercise, apply one or more of the control flow obfuscations available in SandMark and observe their impact by decompiling the obfuscated bytecode using Jad. Show that the Java bytecode reversing solution illustrated in the animated tutorial in section 5.4 can no longer be carried out as demonstrated.

8.4 Animated Solution to the Java Bytecode Antireversing Exercise

For instructional purposes, an animated solution to the exercise in section 8.3 that demonstrates the use of antireversing tools mentioned throughout section 8 to obfuscate the Java Password Vault application was created using Qarbon Viewlet Builder and can be viewed using Macromedia Flash Player. The tutorial begins with the Java Password Vault application, ProGuard, SandMark, Jad, CafeBabe, and Sun's Java JDK already installed on a Windows® XP machine. Figure 8.2 contains an example slide from the animated solution. The animated solution for the Java bytecode antireversing exercise can be downloaded from the following location:

- Java Bytecode Antireversing Animated Solution:
  http://reversingproject.info/repository.php?fileID=8_4_1
Begin viewing the tutorial by extracting `password_vault_java_antireversing_exercise.zip` to a local directory and either running `password_vault_java_antireversing_exercise.exe` which should launch the standalone version of Macromedia Flash Player, or by opening the file `password_vault_java_antireversing_exercise_viewlet_swf.html` in a Web browser.

**Figure 8.2.** Sample slide from the Java antireversing animated tutorial.

9 Conclusion

In this chapter we have covered some of the basic concepts related to reverse engineering and protecting Wintel machine code and Java bytecode. Since many similarities exist between the machine instruction set for different platforms, and Java bytecode can now be generated using other languages such as Ruby and Groovy, these concepts can be useful in a more general context. While the consistent theme throughout the exercises was either the disabling or protection of a trial limitation, which was selected for it's obvious appeal, many more less controversial scenarios can be attempted with the base knowledge gleaned from the exercises. Having learned that it is possible to alter the behavior of machine code or bytecode, one could use this knowledge to fix a bug or even add new function to an application for which the source code is lost. It's no secret that intellectual property is very important to software companies, therefore the
experiences gained from the antireversing exercises can be very helpful in commercial settings, making one a more attractive job candidate, even if they are simply just aware of these issues.

Institutions that employ information technology are always looking for candidates that can help them understand what they have and how it can be evolved to interact with the latest technologies. Engineers can certainly benefit from reverse engineering skills when attempting to help these institutions understand their current technology stack and recommend an integration strategy for new technologies. No less important of course are software security issues such as being able to determine how the latest virus or worm infects computer systems. The detection of viruses and spyware deeply leverages reverse engineering skills by requiring both live and static analysis of machine code and bytecode and attempting to determine malicious code sequences.
References


[16] P. Kouznetsov, Jad v1.5.8g: Jad is a Java decompiler, i.e. program that reads one or more Java class files and converts them into Java source files which can be compiled again. [Online]. Available: http://www.kpdus.com/jad.html. (Accessed: Jun 15, 2008).


