

2009

Software reverse engineering education

Teodoro Cipresso
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Part of the [Software Engineering Commons](#)

Recommended Citation

Cipresso, Teodoro, "Software reverse engineering education" (2009). *Master's Theses*. 3734.
DOI: <https://doi.org/10.31979/etd.4ppy-2cjc>
https://scholarworks.sjsu.edu/etd_theses/3734

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

SOFTWARE REVERSE ENGINEERING EDUCATION

A Thesis

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Teodoro Cipresso

August 2009

UMI Number: 1478574

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1478574

Copyright 2010 by ProQuest LLC.

All rights reserved. This edition of the work is protected against unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

©2009

Teodoro Cipresso

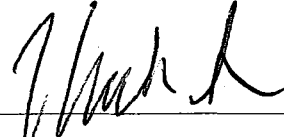
ALL RIGHTS RESERVED

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Thesis Titled
SOFTWARE REVERSE ENGINEERING EDUCATION

by
Teodoro Cipresso

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE



Dr. Mark Stamp,

Department of Computer Science

5/20/09

Date



Dr. David Taylor,

Department of Computer Science

5/20/09

Date



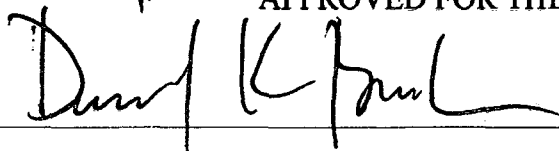
Dr. Robert Chun,

Department of Computer Science

5/20/09

Date

APPROVED FOR THE UNIVERSITY



Associate Dean

Office of Graduate Studies and Research

7/20/09

Date

ABSTRACT

SOFTWARE REVERSE ENGINEERING EDUCATION

by Teodoro Ciproso

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 as well as to protect intellectual property. It became frighteningly apparent during the Y2K crisis that reverse engineering skills were not commonly held amongst programmers. Since that time, much research has been undertaken to formalize the types of activities that fall into the category of reverse engineering so that these skills can be taught to 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 most of the related activities fall into one of two categories: software development related and security related. Hands-on reverse engineering 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.

ACKNOWLEDGEMENTS

I would like to thank Dr. Mark Stamp for his enduring patience as I struggled to flush out the details of this work. I would also like to thank my committee members, Dr. David Taylor and Dr. Robert Chun, for their support in this effort.

Last but not least, I would like to thank my wife Karyn, who has encouraged me throughout my graduate career to persevere through the rough patches, and my cat Freddy, who always kept me company as I typed many suns to sleep.

Table of Contents

1	Introduction.....	1
2	Reverse Engineering in Software Development.....	3
3	Reverse Engineering in Software Security.....	6
4	Reversing and Patching Wintel Machine Code.....	9
4.1	Decompilation and Disassembly of Machine Code.....	11
4.2	Wintel Machine Code Reversing and Patching Exercise.....	14
4.3	Recommended Reversing Tool for the Wintel Exercise.....	15
4.4	Animated Solution to the Wintel Reversing Exercise.....	17
5	Reversing and Patching Java Bytecode.....	20
5.1	Decompiling and Disassembling Java Bytecode.....	21
5.2	Java Bytecode Reversing and Patching Exercise.....	25
5.3	Recommended Reversing Tool for the Java Exercise.....	26
5.4	Animated Solution to the Java Reversing Exercise.....	27
6	Basic Anti-Reversing Techniques.....	29
7	Applying Anti-Reversing Techniques to Wintel Machine Code.....	31
7.1	Eliminating Symbolic Information in Wintel Machine Code.....	31
7.2	Basic Obfuscation of Wintel Machine Code.....	35
7.3	Protecting Source Code Through Obfuscation.....	40
7.4	Advanced Obfuscation of Machine Code.....	42
7.5	Wintel Machine Code Anti-Reversing Exercise.....	44
7.6	Solution to the Wintel Anti-Reversing Exercise.....	44
7.6.1	Encryption of String Literals.....	45
7.6.2	Obfuscating the Numeric Representation of the Record Limit.....	47
7.6.3	Control Flow Obfuscation for the Record Limit Check.....	48
7.6.4	Analysis of the Control Flow Obfuscation Using Run Traces.....	53
8	Applying Anti-Reversing Techniques to Java Bytecode.....	56
8.1	Eliminating Symbolic Information in Java Bytecode.....	58
8.2	Preventing Decompilation of Java Bytecode.....	63
8.3	A Java Bytecode Code Anti-Reversing Exercise.....	68
8.4	Animated Solution to the Java Bytecode Anti-Reversing Exercise.....	69
9	Reengineering and Reuse of Legacy Software Applications.....	70
9.1	Legacy Software Reengineering and Reuse Exercise.....	84
9.2	Legacy Software Reengineering and Reuse Exercise Solution.....	86
10	Identifying, Monitoring, and Reporting Malware.....	98
10.1	Malware Identification and Monitoring Exercise.....	106
10.2	Malware Identification and Monitoring Exercise Solution.....	106
	Conclusion.....	107
	References.....	109

List of Tables

Table 4.1	Result of decompiling HelloWorld.exe using Boomerang.	13
Table 4.2	Quick reference for panes in CPU window of OllyDbg.	16
Table 5.1	Source listing for ListArguments.java.	22
Table 5.2	Java bytecode contained in ListArguments.class.	23
Table 5.3	Jad decompilation of ListArguments.class.	24
Table 7.1	Debugging information inserted into machine code.	33
Table 7.2	Listing of VerifyPassword.cpp and disassembly of VerifyPassword.exe.	36
Table 7.3	Simple substitution cipher used to protect string constants.	38
Table 7.4	VerifyPasswordObfuscated.cpp and corresponding disassembly.	39
Table 7.5	COBF obfuscation results for VerifyPassword.cpp.	41
Table 7.6	Encrypted strings are decrypted each time they are displayed.	45
Table 7.7	Using a function of the record limit to obfuscate the condition.	47
Table 7.8	Implementation of the control flow obfuscation in Fig. 7.3.	51
Table 7.9	Statistical data gathered for randomized control-flow obfuscation.	56
Table 8.1	Unobfuscated source listing of CheckLimitation.java.	60
Table 8.2	Jad decompilation of ProGuard obfuscated bytecode.	61
Table 8.3	Jad decompilation of SandMark (and ProGuard) obfuscated bytecode.	62
Table 8.4	Listing of DateTime.java.	67
Table 8.5	Jad decompilation of DateTime.class obfuscated by Zelix Klassmaster.	68
Table 9.1	Sample business logic component to reuse and reengineer.	76
Table 9.2	Interface data structure SMPLCALC-INTERFACE in SMPLCALC.cpy.	86
Table 9.3	XML Schema generated the from COBOL data structure.	87
Table 9.4	Partial listing of SmplCalcJaxbMarshaller.java interaction with JAXB.	90
Table 9.5	Updates to JSimpleCalculator.java in support of JAXB marshalling.	91
Table 9.6	Example native method declaration for the JNI XML bridge.	92
Table 9.7	Example implementation of the Java to COBOL JNI XML bridge.	93
Table 9.8	Implementation of a COBOL XML layer to the legacy application	95
Table 9.9	Example run of the solution code with debug statements turned on.	96

List of Figures

Figure 2.1	Development process for maintaining legacy software.	4
Figure 2.2	Development related software reverse engineering scenarios.	5
Figure 3.1	Security related software reverse engineering scenarios.	8
Figure 4.1	The five panes of the OllyDbg graphical workbench.	16
Figure 4.2	Sample slide from the machine code reversing animated tutorial.	19
Figure 5.1	Execution of Java bytecode versus machine code.	21
Figure 5.2	FrontEnd Plus workbench session for ListArguments.class.	27
Figure 7.1	Result of obfuscating all string literals in the program.	46
Figure 7.2	Record limit comperands are represented as exponents with a base of 2.	49
Figure 7.3	Obfuscated control flow logic for testing the password record limit.	51
Figure 7.4	Edit-distances between three run traces of the trial limitation check.	55
Figure 8.1	Usage of opaque predicates to prevent decompilation.	65
Figure 8.2	Sample slide from the Java antireversing animated tutorial.	70
Figure 9.1	Layers of a well-structured legacy software application.	73
Figure 9.2	Mapping legacy functional discriminators to an object-oriented design.	75
Figure 9.3	Example JCA implementation for accessing a legacy application.	79
Figure 9.4	Architecture for legacy application reengineering and reuse from Java.	83
Figure 9.5	Console-based Java interface to the legacy COBOL program.	89
Figure 10.1	Process Monitor session for the Password Vault application.	102
Figure 10.2	Example ThreatExpert report summary for submitted malware.	104
Figure 10.3	Console-based UI for the Alarm Clock example software Trojan.	105

1 Introduction

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. The sections that follow 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]. Shortly after the preparation for 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 Business to Consumer (B2C) web sites.

Today's technology is unfortunately tomorrow's legacy system. For example, the Web 2.0 revolution sees the current crop of web sites 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 decrease 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 may have been acquired from a vendor who is no longer 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 are 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. Fig. 2.1 illustrates a process a software engineer might follow when maintaining legacy software systems. 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

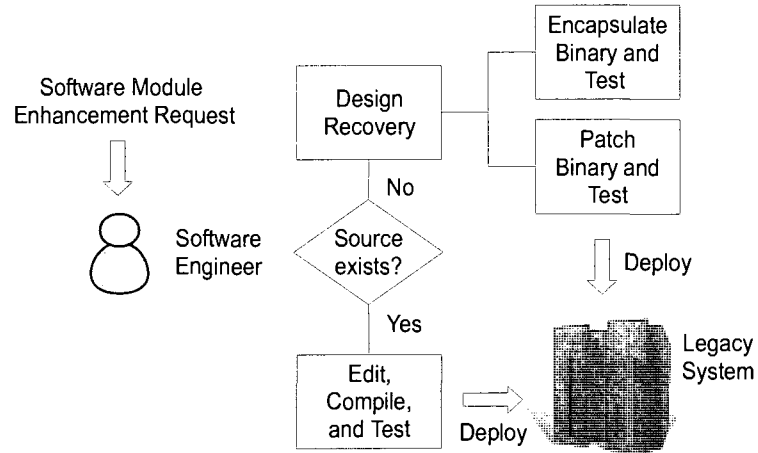


Figure 2.1. Development process for maintaining legacy software.

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 models 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. Fig. 2.2 summarizes the software development related reverse engineering scenarios.

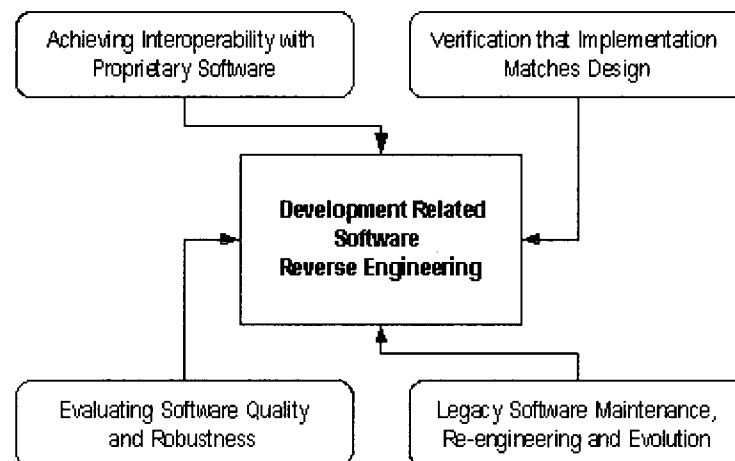


Figure 2.2. Development related software reverse engineering scenarios.

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 transformations to software binaries until it is packaged for shipment to customers. 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 anti-reversing 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 anti-virus 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. Similar to development related reverse engineering—the scenarios cover a broad spectrum of activities: 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. Fig. 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 anti-reversing 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.

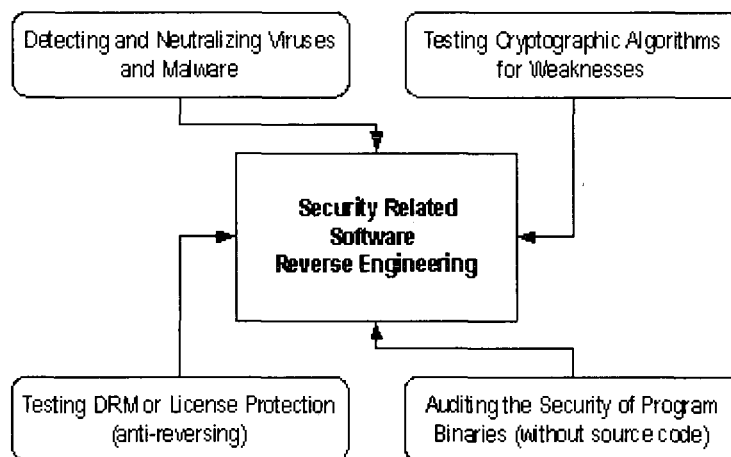


Figure 3.1. Security related software reverse engineering scenarios.

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 executables can be extremely helpful when attempting to reverse engineer machine code. 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 strategy.

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.

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

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

HelloWorld.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):

```
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:
11: // address: 0x401150
12: void procl()
13: {
14:     __size32 eax; // r24
15:     __size32 ebp; // r29
16:     __size32 ebx; // r27
17:     int ecx; // r25
18:     int edx; // r26
19:     int esp; // r28
20:     SetUnhandledExceptionFilter();
21:     proc5(pc, pc, 0x401000, ebx, ebp, eax, ecx, edx, ebx,
        esp - 4, SUBFLAGS32(esp - 44, 4, esp - 48), esp - 48 == 0,
        (unsigned int)(esp - 44) < 4);
22: }
```

[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 you 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 your company’s Web site, you 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]. Fig. 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 Fig. 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.

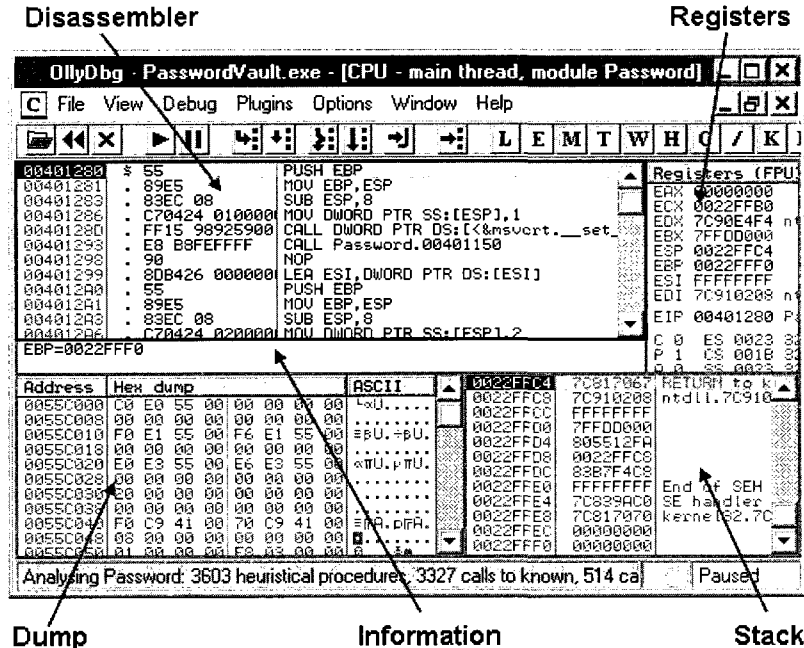


Figure 4.1. The five panes of the OllyDbg graphical workbench.

Table 4.2. Quick reference for panes in CPU window of OllyDbg.

Pane	Capabilities
Disassembler	<ul style="list-style-type: none"> ➤ Edit, debug, test, and patch a binary executable using actions available on a popup menu. ➤ Patch an executable by copying edits to the disassembly back to the binary.
Dump	<ul style="list-style-type: none"> ➤ Display the contents of memory or a file in one of 7 predefined formats: byte, text, integer, float, address, disassembly, or PE Header. ➤ Set memory breakpoints (triggered when a particular memory location is read from or written to). ➤ Locate references to data in the disassembly (executable code).
Information	<ul style="list-style-type: none"> ➤ Decode and resolve the arguments of the currently selected assembly instruction in the Disassembler pane. ➤ Modify the value of register arguments.

Registers	<ul style="list-style-type: none"> ➤ View memory locations referenced by each argument in either the Disassembler or Dump panes. ➤ Decodes and displays the values of the CPU and FPU (Floating-Point Unit) registers for the currently executing thread. ➤ Floating point register decoding can be configured for MMX (Intel) or 3DNow! (AMD) multimedia extensions. ➤ Modify the value of CPU registers.
Stack	<ul style="list-style-type: none"> ➤ Display the stack of the currently executing thread. ➤ Trace stack frames. In general, stack frames are used to: <ul style="list-style-type: none"> • Restore the state of registers and memory on return from a call statement. • Allocate storage for the local variables, parameters, and return value of the called subroutine. • Provide a return address.

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/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. Fig. 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

- *Password Vault C/C++ Source code:*

http://reversingproject.info/repository.php?fileID=4_1_2

- *Password Vault C/C++ Windows® installer:*

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.

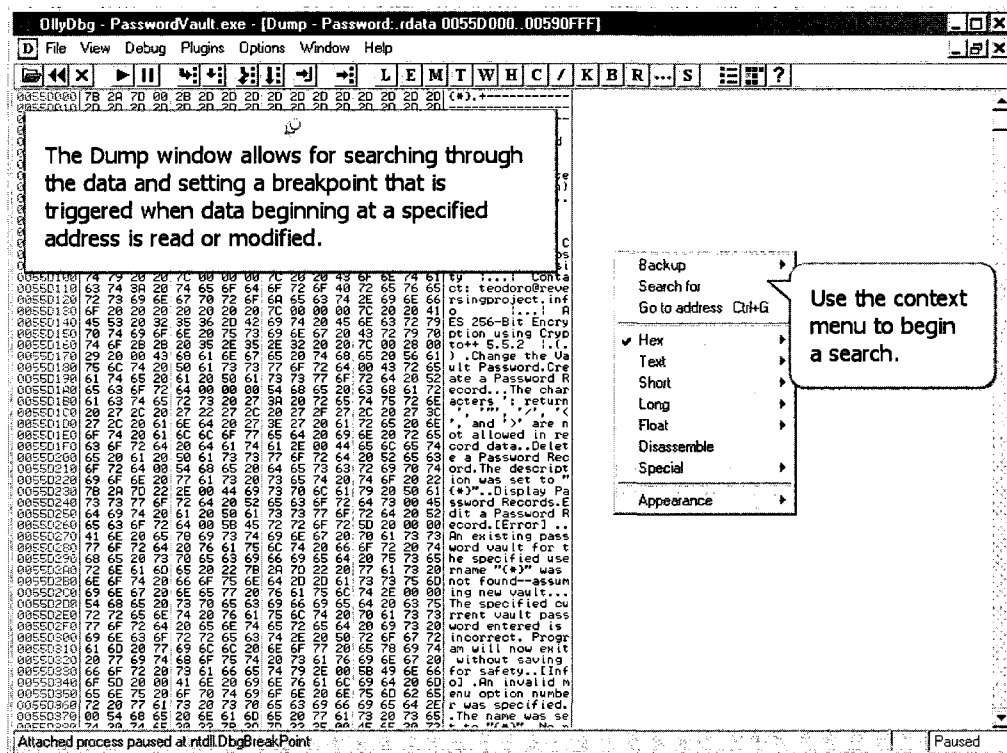


Figure 4.2. Sample slide from the machine code reversing animated tutorial.

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 (Fig. 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 executables built using two different compilers were 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 the pattern *TopLevelClassName.class*.

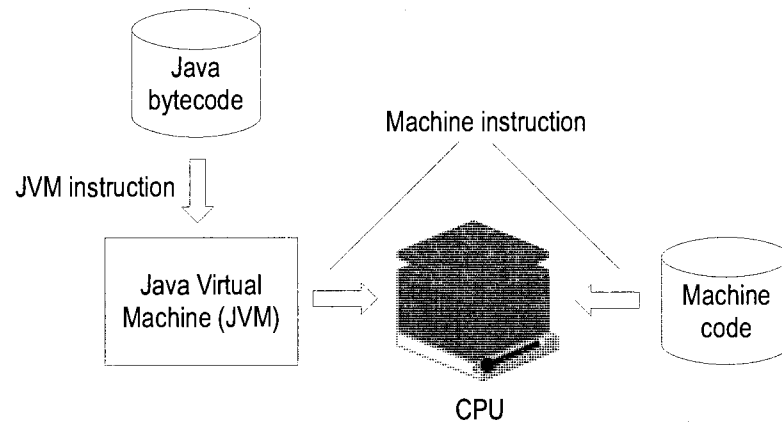


Figure 5.1. Execution of Java bytecode versus machine code.

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/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:
03: public class ListArguments {
04:     public static void main(String[] arguments){
05:         for (int i = 0; i < arguments.length; i++) {
06:             System.out.println("Argument[" + i + "]: " + arguments[i]);
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*.

0:	iconst_0	
1:	istore_1	
2:	iload_1	
3:	aload_0	
4:	arraylength	
5:	if_icmpge	50
8:	getstatic	#2; // java/lang/System.out
11:	new	#3; // java/lang/StringBuilder
14:	dup	
15:	invokespecial	#4; // java/lang/StringBuilder.init
18:	ldc	#5; // "Argument["
20:	invokevirtual	#6; // java/lang/StringBuilder.append
23:	iload_1	
24:	invokevirtual	#7; // java/lang/StringBuilder
27:	ldc	#8; // "]"
29:	invokevirtual	#6; // java/lang/StringBuilder.append
32:	aload_0	
33:	iload_1	
34:	aaload	
35:	invokevirtual	#6; // java/lang/StringBuilder.append
38:	invokevirtual	#9; // java/lang/StringBuilder.toString
41:	invokevirtual	#10; // java/io/PrintStream.println
44:	iinc	1, 1
47:	goto	2
50:	return	

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 directly editing bytecode is difficult, and that disassembling bytecode results in pseudo-assembly which cannot be compiled, it would seem that losing Java source code is more dire of a situation than losing C/C++ source code, but of course this is not the case because, 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 the exercise given in Section 4.2 for Wintel machine code. Imagine that you have just implemented a *Java* version of 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 your company’s Web site, you 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. Fig. 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.

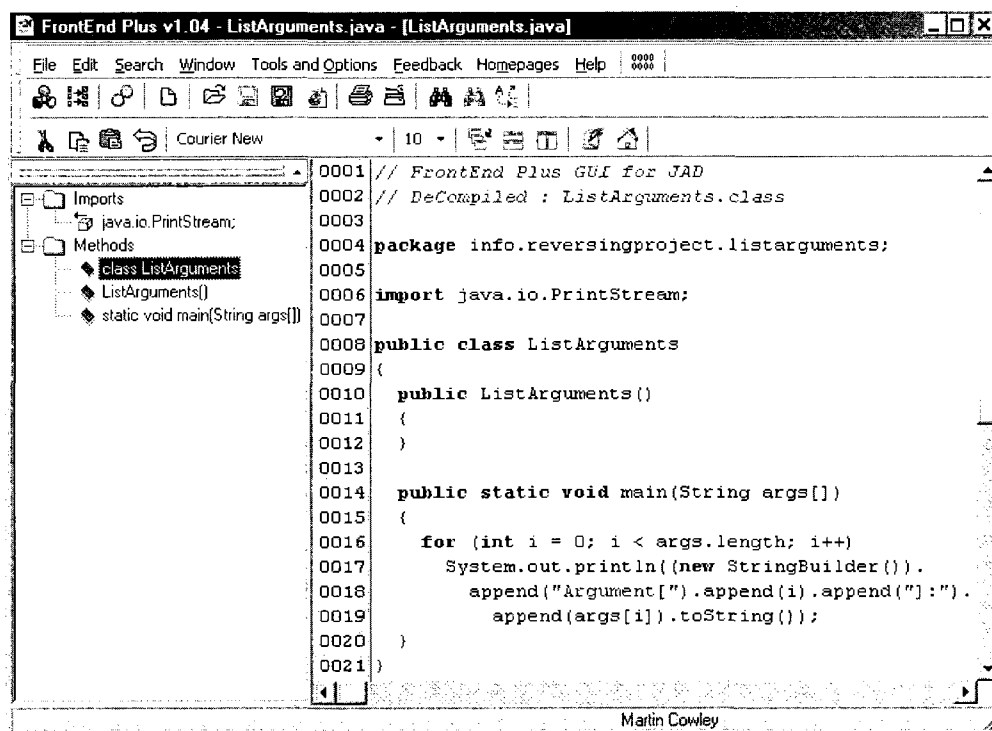


Figure 5.2. FrontEnd Plus workbench session for ListArguments.class.

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, anti-reversing 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. Fig. 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 Anti-Reversing 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 anti-reversing techniques to both implementations in order to make it significantly more difficult for the trial limitation to be disabled. While anti-reversing 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 anti-reversing 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 anti-reversing 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 anti-reversing 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 anti-reversing strategies used.

[5] describes three basic anti-reversing 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 it's 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 Anti-Reversing Techniques to Wintel Machine Code

Extreme care must be taken when applying anti-reversing 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 becomes meaningless [18]. Some of the anti-reversing 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 not be surprised if some compilers have difficulty with obfuscated source code. Following the basic anti-reversing 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 are 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.

Calculator.cpp:

```

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

```

Calculator.s (abbreviated assembly):

```

01: .file "Calculator.cpp"
02: .stabs "C:/SRECD/MiscCPPSource/Calculator/",100,0,0,Ltext0
03: .stabs "Calculator.cpp",100,0,0,Ltext0
04: .stabs "main:F(0,3)",36,0,12,_main
05: .stabs "argc:p(0,3)",160,0,12,8
06: .stabs "argv:p(40,35)",160,0,12,12
06: __main:
07: .stabs "Calculator.cpp",132,0,0,Ltext
08: call __Z5doAddii
09: call __Z5doSubii
10: call __Z5doMulii

```

```

11: .stabs  "_Z5doAddii:F(0,18)",36,0,33,___Z5doAddii
12: .stabs  "op1:p(0,3)",160,0,33,8
13: .stabs  "op2:p(0,3)",160,0,33,12
14: ___Z5doAddii:
15:  movl    12(%ebp), %eax
16:  addl    8(%ebp), %eax
17: .stabs  "_Z5doSubii:F(0,18)",36,0,34,___Z5doSubii
18: .stabs  "op1:p(0,3)",160,0,34,8
19: .stabs  "op2:p(0,3)",160,0,34,12
20: ___Z5doSubii:
21: .stabsn 68,0,34,LM33-___Z5doSubii
22:  movl    8(%ebp), %eax
23:  subl    %edx, %eax
24: .stabs  "_Z5doMulii:F(0,18)",36,0,35,___Z5doMulii
25: .stabs  "op1:p(0,3)",160,0,35,8
26: .stabs  "op2:p(0,3)",160,0,35,12
27: ___Z5doMulii:
28: .stabsn 68,0,35,LM35-___Z5doMulii
29:  movl    8(%ebp), %eax
30:  imull   12(%ebp), %eax

```

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. 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:

```
01: int main(int argc, char *argv[])
02: {
03:     const char *password = "jup!ter";
04:     string specified;
05:     cout << "Enter password: ";
```


10, we can obscure the test by checking if $1.2^a < 1.2^{10}$ 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.

SubstitutionCipher.h:

```
01: class SubstitutionCipher
02: {
03: public:
04:     SubstitutionCipher();
05:     string encryptToHex(string plainText);
06:     string decryptFromHex(string cipherText);
07: private:
08:     unsigned char encryptTable[256];
09:     unsigned char decryptTable[256];
10:     char hexByte[2];
11: };
```

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 corresponding disassembly.

VerifyPasswordObfuscated.cpp:

```
01: #include "substitutioncipher.h"
02: using namespace std;
03: static const char *password = "77827D2E81727F";
04: static const char *enter_password = "527B81727F2D7D6E8080847C
    7F71472D";
05: static const char *password_ok = "685C586A2D4E70707280802D747
    F6E7B8172713B";
06: static const char *password_bad = "68527F7F7C7F6A2D4E70707280
    802D71727B7672713B";
07: int main(int argc, char *argv[])
08: {
09:     SubstitutionCipher cipher;
10:     string specified;
11:     cout << cipher.decryptFromHex(enter_password);
12:     getline(cin, specified);
13:     if (specified.compare(cipher.decryptFromHex(password)) == 0)
14:     {
15:         cout << cipher.decryptFromHex(password_ok) << endl;
16:     } else
17:     {
```

```

18:      cout << cipher.decryptFromHex(password_bad) << endl;
19:  }
20: }

```

VerifyPasswordObfuscated.exe disassembly (abbreviated):

.RDATA SECTION

```

00445000 35323742383137323746324437443645 527B81727F2D7D6E
00445010 38303830383437433746373134373244 8080847C7F71472D
00445020 00373738323744324538313732374600 .77827D2E81727F.
00445030 36383543353836413244344537303730 685C586A2D4E7070
00445040 37323830383032443734374636453742 7280802D747F6E7B
00445050 38313732373133420000000036383532 8172713B....6852
00445060 37463746374337463641324434453730 7F7F7C7F6A2D4E70
00445070 37303732383038303244373137323742 707280802D71727B
00445080 37363732373133420000000000000000 7672713B.....

```

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 anti-reversing 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 client's 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\x21\x74
03: \x65\x72"; lh la; lb<<"\x45\x6e\x74\x65\x72\x20\x70\x61\x73\x73
04: \x77\x6f\x72\x64""\x3a\x20"; li(lq, la); lm(la.lg(lc)==0){lb<<"\x5b
05: \x4f\x4b\x5d\x20\x41" "\x63\x63\x65\x73\x73\x20\x67\x72\x61\x6e
06: \x74\x65\x64\x2e"<<le; }lr{lb<<"\x5b\x45\x72\x72\x6f\x72\x5d
07: \x20\x41\x63\x63\x65\x73\x73\x20\x64" "\x65\x6e\x69\x65
08: \x64\x2e"<<le; }}
```

COBF generated header (cobf.h):

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

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 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 destroy the high-level language structure of a program. For example, if a programmer used the structured programming technique of functional decomposition, inlining the code of many functions into a single function in the machine code would make it impossible to recover the original program structure. 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 an 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 during execution of the obfuscated code.

7.5 Wintel Machine Code Anti-Reversing Exercise

Apply the anti-reversing 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 and patching animated solution in Section 4.4 can no longer be carried out as demonstrated.

7.6 Solution to the Wintel Anti-Reversing Exercise

The solution to the Wintel machine code anti-reversing exercise is given through comparisons of the original and obfuscated source code of the Password Vault application. As each anti-reversing 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 anti-reversing transformations have been applied, the impact they have on the machine code and how reversing the Password Vault application becomes more difficult is covered; 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 *password_vault_cpp_obfuscated* 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.

```
-----
133 case __createPasswordRecord: return  "Create a Password Record";
    ==> 137 case __createPasswordRecord:
        DecryptMessageText ("507F726E81722D6E2D5D6E8080847C7F712D5F72707C7F7
        1", _textBuffer);
-----

186 case __recordLimitReached: return  "Thank you for trying Password
Vault! You have reached the maximum number of records allowed in this
trial version.";
    ==> 190 case __recordLimitReached:
        DecryptMessageText ("61756E7B782D867C822D737C7F2D817F86767B742D5D6E8
        080847C7F712D636E8279812E2D667C822D756E83722D7F726E707572712D817572
        2D7A6E85767A827A2D7B827A6F727F2D7C732D7F72707C7F71802D6E79797C84727
        12D767B2D817576802D817F766E792D83727F80767C7B3B", _textBuffer);
-----

205 void PasswordVaultConsoleUtil::DecryptMessageText(const char
*_cipherText, string *_plainTextBuffer)
206 {
208     string cipherText(_cipherText);
210     SubstitutionCipher cipher;
212     _plainTextBuffer->assign(cipher.decryptFromHex(cipherText));
214 }
-----
```

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

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!

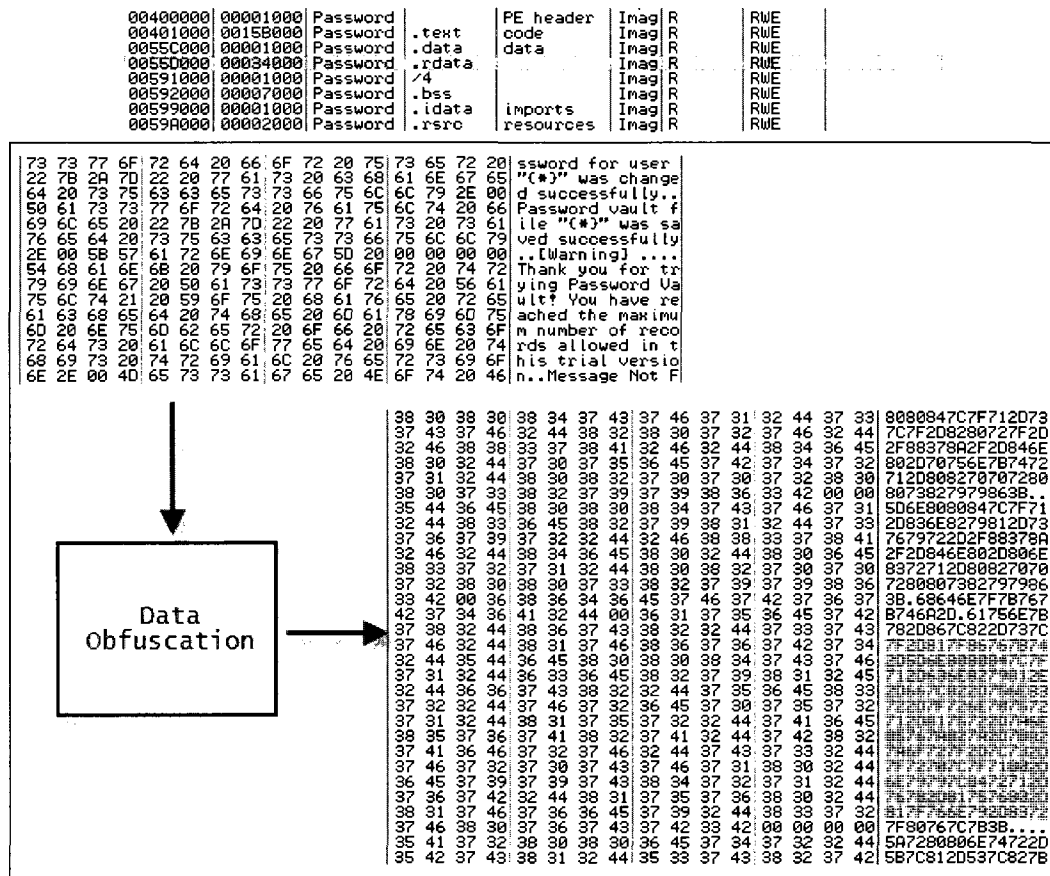


Figure 7.1. Result of obfuscating all string literals in the program.

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 α 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. Using a function of the record limit to obfuscate the condition.

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

The effects of the source code changes in Table 7.7 on the machine code are shown in Fig. 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 Fig. 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.

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

```

004070E0 83BD 00FFFFFF 04      CMP DWORD PTR SS:[EBP-100],4
004070E7 76 21              JBE SHORT Password.0040710A
004070E9 C74424 08 00000000    MOV DWORD PTR SS:[ESP+8],0
004070F1 C74424 04 00000000    MOV DWORD PTR SS:[ESP+4],0
004070F9 C70424 36000000      MOV DWORD PTR SS:[ESP],36
00407100 E8 01A3FFFF          CALL Password.004070C4
00407105 E9 BA070000          JMP Password.004070C4
0040710A 8045 08              LEA EAX,DWORD PTR SS:[EBP-28]
0040710D 890424              MOV DWORD PTR SS:[ESP],EAX
00407110 C785 08FFFFFF FFFFFFFF MOV DWORD PTR SS:[EBP-F8],-1
0040711A E8 3BBCFFFF          CALL Password.00402D5A

```

```
if ((pow(2.0, (double)passwordStore.
getRecords().size()) >= pow(2.0, 5.0)))
```

Computation
Obfuscation

```

00407100 DD05 20E45500        FLD QWORD PTR DS:[55E420]
00407106 DD85 F8FEFFFF        FLD QWORD PTR SS:[EBP-108]
0040710C DAE9                FUCOMPP
0040710E DFE0                FSTSW AX
00407110 9E                SAHF
00407111 73 02              JNB SHORT Password.00407115
00407113 EB 2B              JMP SHORT Password.00407140
00407115 C74424 08 00000000    MOV DWORD PTR SS:[ESP+8],0
0040711D C74424 04 00000000    MOV DWORD PTR SS:[ESP+4],0
00407125 C70424 36000000      MOV DWORD PTR SS:[ESP],36
0040712C C785 08FFFFFF FFFFFFFF MOV DWORD PTR SS:[EBP-F8],-1
00407136 E8 CBA2FFFF          CALL Password.004070C4
0040713B E9 BA070000          JMP Password.004070C4
00407140 8045 08              LEA EAX,DWORD PTR SS:[EBP-28]
00407143 890424              MOV DWORD PTR SS:[ESP],EAX
00407146 C785 08FFFFFF FFFFFFFF MOV DWORD PTR SS:[EBP-F8],-1
00407150 E8 05BCFFFF          CALL Password.00402D5A

```

Live analysis of the computation

```

00407100 DD05 20E45500        FLD QWORD PTR DS:[55E420]
00407106 DD85 F8FEFFFF        FLD QWORD PTR SS:[EBP-108]
0040710C DAE9                FUCOMPP
0040710E DFE0                FSTSW AX
00407110 9E                SAHF
00407111 73 02              JNB SHORT Password.00407115
00407113 EB 2B              JMP SHORT Password.00407140

DS:[0055E420]=32.00000000000000
Stack SS:[0022FBA0]=32.00000000000000

00407100 DD05 20E45500        FLD QWORD PTR DS:[55E420]
00407106 DD85 F8FEFFFF        FLD QWORD PTR SS:[EBP-108]
0040710C DAE9                FUCOMPP
0040710E DFE0                FSTSW AX
00407110 9E                SAHF
00407111 73 02              JNB SHORT Password.00407115
00407113 EB 2B              JMP SHORT Password.00407140

ST(1)=32.0000000000000000
ST=32.0000000000000000

```

The record limit of 5 is obscured by the use of the value 32.0 (2^5) when the operands are loaded and the condition is tested.

Figure 7.2. Record limit comparands 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* is 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.

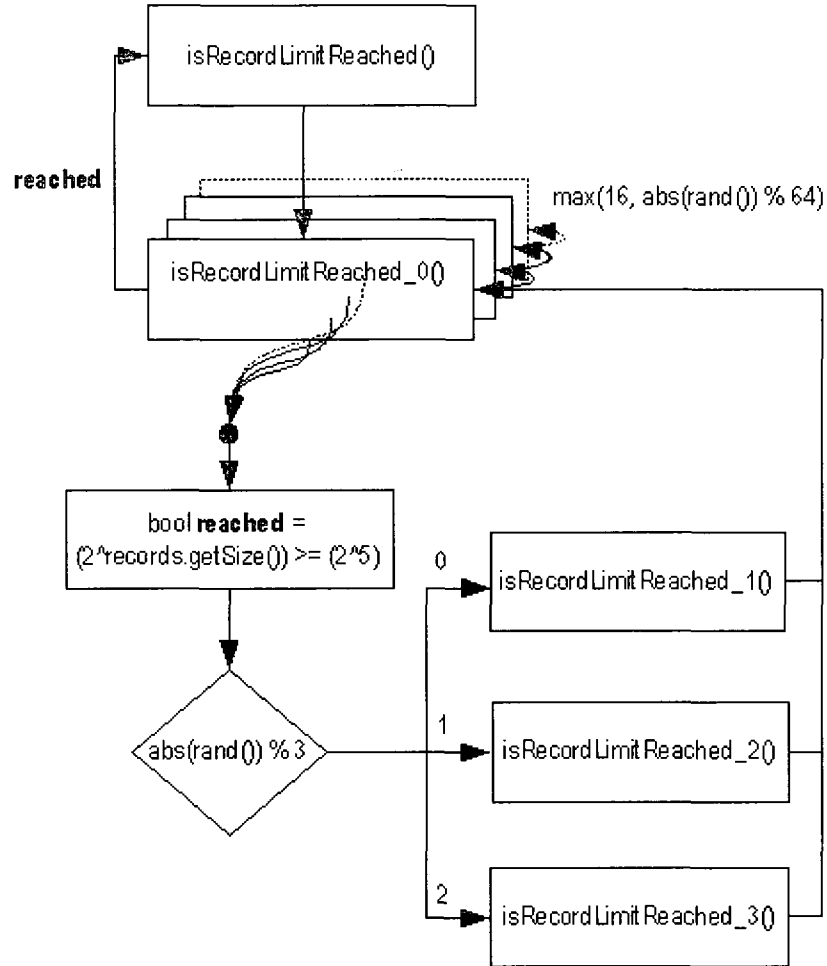


Figure 7.3. Obfuscated control flow logic for testing the password record limit.

Table 7.8. Implementation of the control flow obfuscation in Fig. 7.3.

PasswordVault.cpp:

```

-----
if (passwordStore.getRecords().size() >= TRIAL_RECORD_LIMIT)
==> if (isRecordLimitReached())
-----

```

```

01: bool PasswordVault::isRecordLimitReached()
02: {
03:     srand(time(NULL));
04:     controlFlowAltRemain = max(4, abs(rand()) % 64);
05:     return isRecordLimitReached_0();
06: }

```

```

07:
08: bool PasswordVault::isRecordLimitReached_0()
09: {
10:     while (controlFlowAltRemain > 0)
11:     {
12:         controlFlowAltRemain--;
13:         isRecordLimitReached_0();
14:     }
15:
16:     bool reached = (pow(2.0,
(double)passwordStore.getRecords().size()) >= pow(2.0, 5.0));
17:
18:     randCallSum = 0;
19:
20:     switch (abs(rand()) % 3)
21:     {
22:     case 0:
23:         randCallSum += isRecordLimitReached_1();
24:         break;
25:     case 1:
26:         randCallSum += isRecordLimitReached_2();
27:         break;
28:     case 2:
29:         randCallSum += isRecordLimitReached_3();
30:         break;
31:     }
32:
33:     return reached;
34: }
35:
36: unsigned int PasswordVault::isRecordLimitReached_1()
37: {
38:     return abs(rand());
39: }
40:
41: unsigned int PasswordVault::isRecordLimitReached_2()
42: {
43:     return abs(rand());
44: }
45:
46: unsigned int PasswordVault::isRecordLimitReached_3()
47: {
48:     return abs(rand());
49: }

```

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. Fig. 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 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

Code path: obfuscated trial limitation check

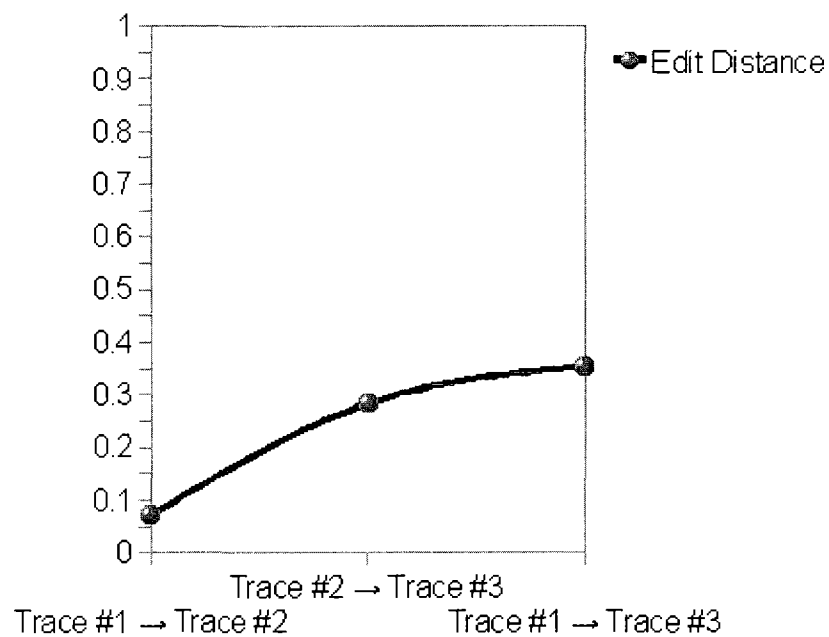


Figure 7.4. Edit-distances between three run traces of the trial limitation check.

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

Trace Comparison	Edit Distance
Trace #1 → Trace #2	101414
Trace #2 → Trace #3	67590
Trace #1 → Trace #3	168892
<i>Mean Edit Distance</i>	<i>112632</i>

Trace Comparison	Standard Deviation
Trace #1 → Trace #2	7932.32
Trace #2 → Trace #3	31849.5
Trace #1 → Trace #3	39781.83

8 Applying Anti-Reversing 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 anti-reversing 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 the 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 0xCAFEBAFE 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. 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,

Table 8.1. Unobfuscated source listing of CheckLimitation.java.

```
01: public class CheckLimitation {
02:
03:     private static int MAX_PASSWORDS = 5;
04:     private ArrayList<String> passwords;
05:
06:     public CheckLimitation()
07:     {
08:         passwords = new ArrayList<String>();
09:
10:         public boolean addPassword(String password)
11:         {
12:             if (passwords.size() >= MAX_PASSWORDS)
13:             {
14:                 System.out.println("[Error] The maximum number of passwords
has been exceeded!");
15:                 return false;
16:             } else
17:             {
18:                 passwords.add(password);
19:                 System.out.println("[Info] password (" + password + ")
added successfully.");
20:                 return true;
21:             }
22:         }
23:
24:         public static void main(String[] arguments)
25:         {
26:             CheckLimitation store = new CheckLimitation();
27:             boolean loop = true;
28:             for (int i = 0; i < arguments.length && loop; i++)
29:                 if (!store.addPassword(arguments[i])) loop = false;
30:         }
31:
32: }
```

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

Table 8.2. Jad decompilation of ProGuard obfuscated bytecode.

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

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.

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 *CheckLimitation.java* bytecode that was first obfuscated using ProGuard and subsequently obfuscated using the “String Encoder” functionality in SandMark.

Table 8.3. Jad decompilation of SandMark (and ProGuard) obfuscated bytecode.

```

01: public class CheckLimitation {
02:
03:     private static int a = 5;
04:     private ArrayList b;
05:
06:     public CheckLimitation()
07:     {
08:         b = new ArrayList();
09:     }
10:
11:     public boolean a(String arg0)
12:     {
13:         if(b.size() >= a)
14:         {
15:             System.out.println(Obfuscator.DecodeString("\253\315\253\315\uFF9E\u2A3
Du5D69\u2AA5\u3884\u91CF\u5341\u5604\uDF5B\uA902\uB6C8\u0C8E\u6761\u1F3
5\u359D\uBD96\uADA4\u946F\u85EE\uE8A0\u9274\u5867\u2C9F\u3077\u5E67\u2A
0B\u90D2\uB839\u58FC\uBE95\u0EBA\uDDF4\u313C\uB751\uFA9D\u166C\u42A3\u6
D1D\uB25A\uA15E\u026E\u6ECE\u908C\u557B\u6ABD\uC5D5\u800C\uD38A\u3D97\u
FB5E\uC4C2\uBBAC\u9ADC\u253E\u769E\u4D32\u4FB3\u0CC7"));
16:             return false;
17:         } else
18:         {
19:             b.add(arg0);
20:             System.out.println((new
StringBuilder()).append(Obfuscator.DecodeString("\253\315\253\315\uFF9E
\u2A31\u5D75\u2AB1\u3884\u91E0\u533C\u5654\uDF6E\uA919\uB6DE\u0CD9\u676
3\u1F26\u3581\uBDDF\uADE1")).append(arg0).append(Obfuscator.DecodeStrin
g("\253\315\253\315\uFFEC\u2A58\u5D7A\u2AB3\u388F\u91D8\u5378\u5604\uDF
7C\uA91F\uB6CE\u0CCD\u6769\u1F27\u3596\uBD99\uADBC\u9476\u85EF\uE8F9\u9

```

```

234")) .toString());
21:     return true;
22: }
23: }
24:
25: public static void main(String arg0[])
26: {
27:     CheckLimitation checklimitation = new CheckLimitation();
28:     boolean flag = true;
29:     for(int i = 0; i < arg0.length && flag; i++)
30:         if(!checklimitation.a(arg0[i])) flag = false;
31: }
32: }

```

Note that each string literal is decrypted using the *Obfuscator* class which was generated by SandMark. Since *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 programmatically 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 Decompile 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. Fig. 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.).

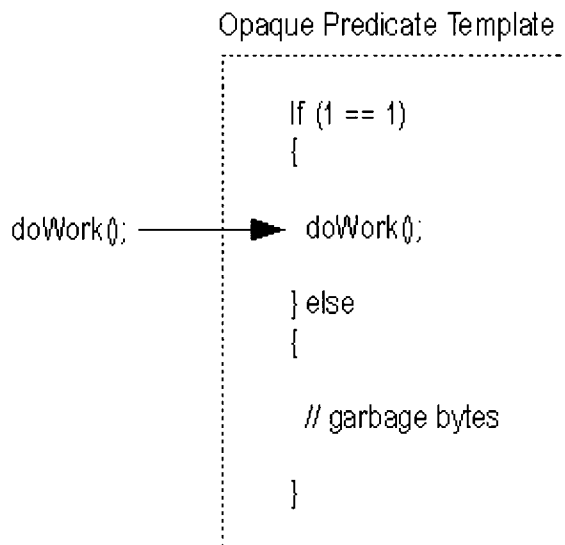


Figure 8.1. Usage of opaque predicates to prevent decompilation.

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.

SandMark's approach of using control flow obfuscations that leverage opaque predicates in an attempt to confuse decompilers 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

Table 8.4. Listing of `DateTime.java`

Listing of `DateTime.java` (abbreviated):

```
01: public static void main(String arguments[])
02: {
03:     new DisplayDateTime().doDisplayDateTime();
04: }
05:
06: public void doDisplayDateTime()
07: {
08:     Date date = new Date();
09:     System.out.println(String.format(DATE_TIME_MASK,
10: date.toString()));
10: }
```

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 anti-reversing 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):

```
01: public class a
02: {
03:     public static void main(String as[])
04:     {
05:         (new a()).a();
06:     }
07:
08:     public void a()
09:     {
10:         boolean flag = c;
11:         Date date = new Date();
12:         System.out.println(String.format(a, new Object[] {
13:             date.toString()}));
14:         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 Anti-Reversing Exercise

Use Java bytecode anti-reversing tools such as ProGuard, SandMark, and CafeBabe on the Java version of the Password Vault application to apply the anti-reversing 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 Wintel machine code anti-reversing 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 Anti-Reversing Exercise

For instructional purposes, an animated solution to the exercise in Section 8.3 that demonstrates the use of anti-reversing tools introduced in Section 8 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. Fig. 8.2 contains an example slide from the animated solution. The animated solution for the Java bytecode anti-reversing exercise can be downloaded from the following location:

➤ *Java Bytecode Anti-Reversing 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.

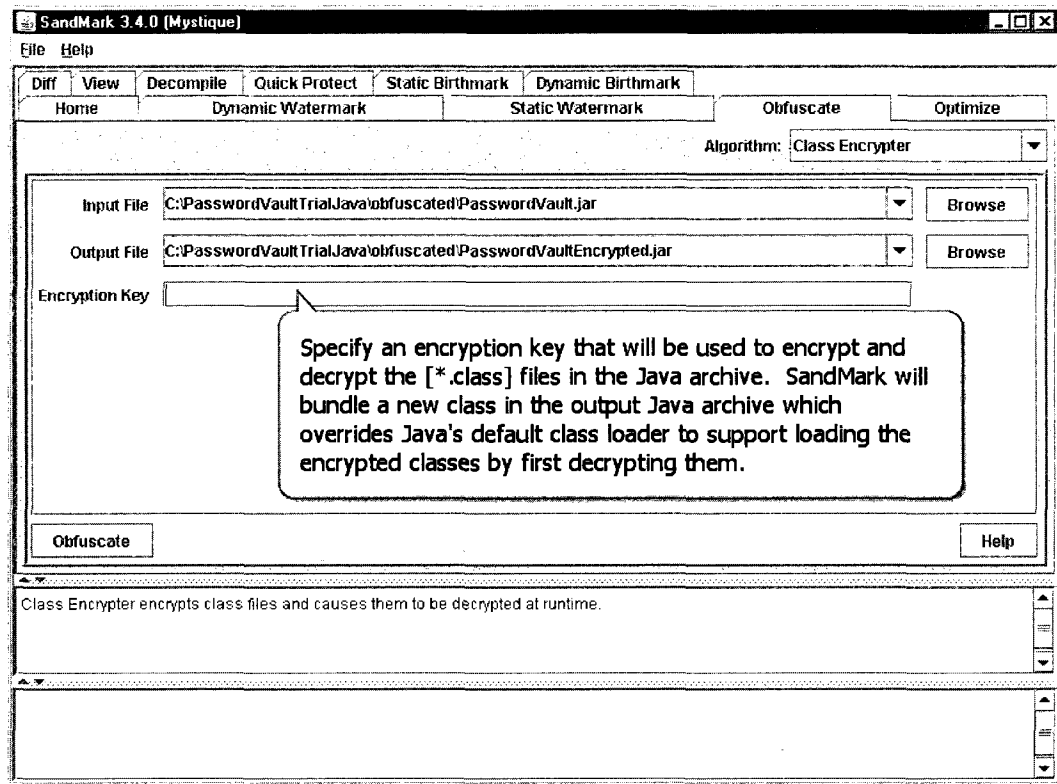


Figure 8.2. Sample slide from the Java anti-reversing animated tutorial.

9 Reengineering and Reuse of Legacy Software Applications

As stated in the introduction, the literature points to a future where the standard approach of “forward engineering” of software will be complimented with reverse engineering to rediscover the architecture and design as the actual implementation is being created. While any application that is greater than five years old can be considered “legacy”, in this section we assume a more severe condition where enough time has passed such that an application has been enhanced and modified by several programmers, over several years, who have since moved on. Most computer science programs of study include object-oriented programming theory; this includes learning how to create

diagrams that illustrate the components of a system as well as their interactions during execution. The hope is that these diagrams will be literally translated in to program code, with a perfect correlation between the envisioned system and the implementation. For small projects, it might be fairly easy to check for consistency between the envisioned design, and what has been implemented, but this is not likely so for large projects. When reverse engineering is continuously used during software development, the information gained could be used to update the design diagrams at all levels of granularity [2]. The challenge here is for the computer programmer to interpret the information gathered from these reverse engineering tools. This will require the programmer to draw upon a skill-set that ranges from low-level system concepts to high-level design. Unfortunately, the future offers little help in undoing the mistakes of the past.

The problem of identifying concrete, reusable components within a software system is especially difficult because as [7] states, “engineers do not know how to design and build truly modular systems from scratch, let alone when starting from legacy code.” In [7], Weide and Hollingsworth's main thesis is that while reverse engineering of legacy software is inherently intractable, some of us will inevitably find ourselves in a situation where no other option is available because the cost of rewriting a large, complex software system is prohibitive. In addition, should one choose to absorb the cost of rewriting a system from scratch, there are no known software development techniques that can guarantee a newly-designed system that will not need to be reengineered at some point down the road.

The question of whether to reengineer or reuse components of a software system most often arises in the context of large business or government organizations. Over time the processes and procedures of a business or organization will inevitably be reflected in the software systems that enable efficient, day-to-day operations [32]. Therefore, it is not possible to change processes and procedures without adjusting or enhancing the software systems that implement them. If good development practices were followed, a legacy software application is typically composed of three layers [32]:

- *Presentation Layer*: components of a software system that accept input and generate output using various types of hardware devices. Input and output can be entered or analyzed by a human or another by another program.
- *Business Logic Layer*: implementation of some subset of the processes and procedures of the business or organization that is relevant to the application. It is unusual for the business logic of one application to implement all of the processes and procedures. For example, the order processing and payroll applications are not likely to have much business logic in common.
- *Data Access Layer*: this layer is responsible for servicing requests to store or retrieve data on behalf of the presentation and business logic layers. The nature of the code in this layer varies depending on the database technology being used. Technology choices range from simple sequential files to industrial-strength relational or hierarchical databases.

Fig. 9.1 illustrates the program architecture one would hope was used when looking to update, reengineer, or reuse a legacy software application.

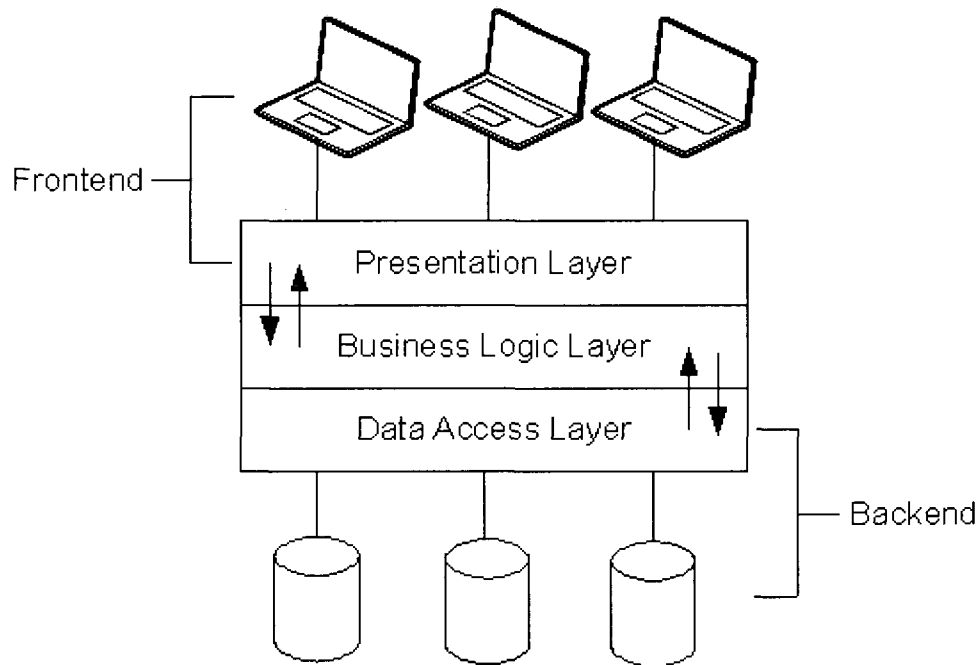


Figure 9.1. Layers of a well-structured legacy software application.

Legacy applications that are not sufficiently componentized, such that their general organization resembles the three layers, are not good candidates for reengineering and reuse. More often than not, most software development projects in business are done under fairly aggressive time constraints, therefore it is not uncommon to find an interleaving of the layers—business logic in the presentation logic, and data access logic in the business logic. The most widely accepted technique to reuse legacy application components is that of *Wrapping* [32], where a new piece of code provides an interface to a legacy application component or layer without requiring code changes to it. This

technique is employed even when the complete source code of a legacy application is available for several reasons: (1) the number of lines of code in any one component or layer is extensive and poorly documented—making the cost of understanding the code well enough to make changes too high, (2) modifying the legacy application to be reusable without a wrapper would require locating all of the application's dependencies so that it can be recompiled and tested (3) application modernization, where a non-traditional interface to the application such as XML in the case of Web or RESTful services is desired.

Creating a wrapper to a legacy machine code application can be quite challenging—especially if all of the source code for the application has been lost. Unless enough of an application's source code remains such that it's possible to identify the names of reusable entry points (procedures) and their I/O data structures, attempting to reuse the application is haphazard at best. While it is possible to learn the names of entry points that have been explicitly exported by an application in the case of a DLL, the names don't indicate the layout of the expected I/O data structures. Probably the best way to discover the entry points and I/O data structures in legacy machine code is to read the source code of other applications which depend on it. For example, if a program α calls procedure θ of program β passing an I/O data structure δ , and α produces correct results, there is good reason to believe that procedure θ in program β can be reused by a third program ρ using signature δ .

The COBOL programming language is most often associated with legacy

software applications. Typically, COBOL programs have a single entry point, which makes the process of identifying reusable methods all but unnecessary because, instead of declaring multiple entry points, it is general practice for legacy COBOL programs to include functional discriminators in their I/O data structure(s) that indicate the desired action(s) to be taken by the program on behalf of the caller. For example, a field called “TRANSACTION-TYPE” with the possible values “DEP”, “WTH”, and “BAL” would serve the same purpose in a COBOL program as the methods “doDeposit(δ)”, “doWithdrawl(δ)“, and “getBalance(δ)” would serve in a Java program.

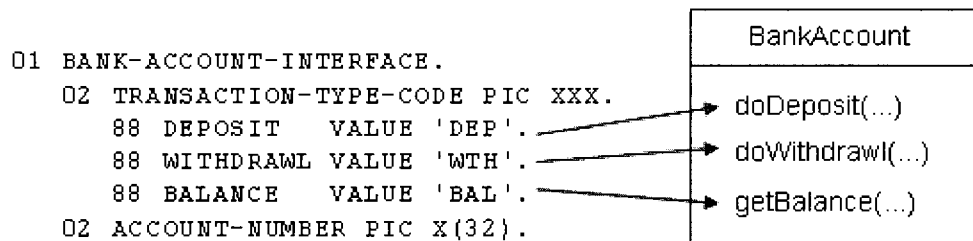


Figure 9.2. Mapping legacy functional discriminators to an object-oriented design.

Fig. 9.2 illustrates how a functional discriminator in a legacy COBOL data structure maps to modern programming strategies such as object-oriented design.

In a real-world situation, we would be looking to reuse legacy components whose machine code is the result of thousands of lines of high-level language statements (COBOL) that implement a particular business process. Instead of going through the error-prone process of rewriting the legacy component, which is likely decades old, we wish to reuse and reengineer it so that it is easily consumed by modern programs and

interfaces. Since our focus is more on reuse and reengineering of legacy code at a basic level, it's not necessary to encumber ourselves with a very large program in order to learn strategies for reuse and reengineering. Therefore, for purposes of demonstration, an example COBOL program *SMPLCALC.cbl*, which implements a simple calculator for integer-only arithmetic, was written to simulate a potentially useful component found in the business logic layer of a legacy business application. The source code for *SMPLCALC.cbl* is given Table 9.1; the program has single entry point that operates on the I/O data structure *SMPCALC-INTERFACE*.

Table 9.1. Sample business logic component to reuse and reengineer.

```

01: *****
02: ** Simple COBOL program that performs integer arithmetic      **
03: *****
04: IDENTIFICATION DIVISION.
05:   PROGRAM-ID. 'SMPLCALC'.
06: DATA DIVISION.
07: WORKING-STORAGE SECTION.
08:   77 MSG-NUMERIC-OVERFLOW PIC X(25)
09:     VALUE 'Numeric overflow occurred'.
10:   77 MSG-SUCCESSFUL PIC X(22)
11:     VALUE 'Completed successfully'.
12: LINKAGE SECTION.
13: * Input/Output data structure
14: 01 SMPLCALC-INTERFACE.
15:   02 SI-OPERAND-1 PIC S9(9) COMP-5.
16:   02 SI-OPERAND-2 PIC S9(9) COMP-5.
17:   02 SI-OPERATION PIC X.
18:     88 DO-ADD VALUE '+'.
19:     88 DO-SUB VALUE '-'.
20:     88 DO-MUL VALUE '*'.
21:   02 SI-RESULT PIC S9(18) COMP-3.
22:   02 SI-RESULT-MESSAGE PIC X(128).
23: PROCEDURE DIVISION USING
24:   BY REFERENCE SMPLCALC-INTERFACE.
25: MAINLINE SECTION.
26: * Perform requested arithmetic
27:   INITIALIZE SI-RESULT SI-RESULT-MESSAGE
28:   EVALUATE TRUE
29:     WHEN DO-ADD

```

```

30:          COMPUTE SI-RESULT = SI-OPERAND-1 + SI-OPERAND-2
31:          ON SIZE ERROR
32:            PERFORM HANDLE-SIZE-ERROR
33:          END-COMPUTE
34:        WHEN DO-SUB
35:          COMPUTE SI-RESULT = SI-OPERAND-1 - SI-OPERAND-2
36:          ON SIZE ERROR
37:            PERFORM HANDLE-SIZE-ERROR
38:          END-COMPUTE
39:        WHEN DO-MUL
40:          COMPUTE SI-RESULT = SI-OPERAND-1 * SI-OPERAND-2
41:          ON SIZE ERROR
42:            PERFORM HANDLE-SIZE-ERROR
43:          END-COMPUTE
44:        END-EVALUATE
45:      * Successful return
46:      MOVE MSG-SUCCESSFUL TO SI-RESULT-MESSAGE
47:      MOVE 2 TO RETURN-CODE
48:      GOBACK
49:      .
50: *****
51: ** Handle numeric overflow and end the program          **
52: *****
53: HANDLE-SIZE-ERROR.
54:   MOVE MSG-NUMERIC-OVERFLOW TO SI-RESULT-MESSAGE
55:   MOVE 16 TO RETURN-CODE
56:   GOBACK
57:   .
58: END PROGRAM 'SMPLCALC' .

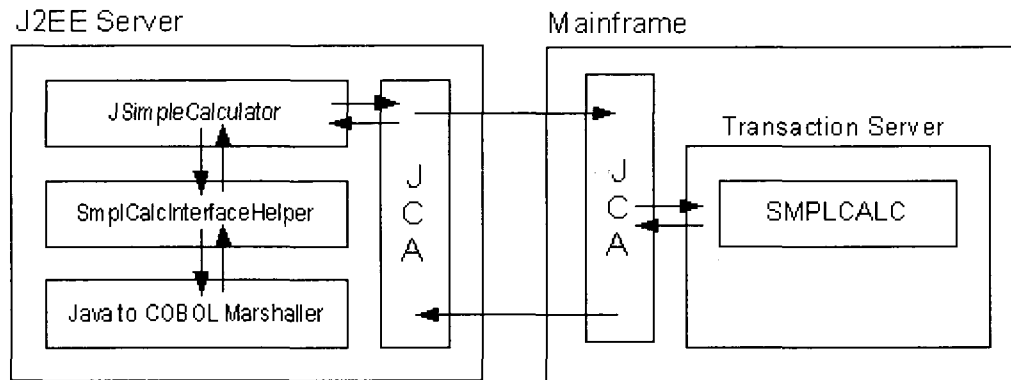
```

Looking at the source code for the COBOL program *SMPLCALC.cbl*, we can easily determine the entry point name and the data layout of the I/O data structure.

However, even knowing the full details of the application's interface does not solve the problem of making it easily reusable from Java or C because of the differences in the language data type systems. For example, *Packed Decimal (Computational-3)* is a numeric type that is commonly found in COBOL mainframe programs, but is not directly supported in the Java and C/C++ languages. Even floating-point numbers can be problematic because some COBOL compilers, including IBM's, do not use the standard

IEEE floating point representation; they instead use decimal floating point [44]. Without detailing all the differences between the COBOL, Java, and C/C++ data models, it suffices to say that writing custom code to convert between COBOL's data model and the language we wish to invoke it from is error-prone and tedious and there are better alternatives.

The problem of mating disparate data models so that new programs, written in modern languages, can interact with legacy software systems, is far from new. There are many commercial tools on the market that can import a COBOL data structure and generate Java helper classes that a programmer can use to build to meet the legacy binary interface using familiar getters and setters. A great many of these tools, including IBM's Rational Application Developer (RAD) [33], leverage Sun Microsystems J2EE Connector Architecture (JCA) [34] to provide a tightly coupled integration between a Java application running in a J2EE container (server) and an enterprise application (likely written in COBOL or PL/I) running on a mainframe. The JCA architecture requires a good deal of middleware to exist between a calling Java application running in the J2EE container and the COBOL application running on a legacy software system. While this middleware is powerful because of its capability to marshall Java data into COBOL and PL/I data, it cannot easily be reused for a local scenario where no server runtimes are involved. Fig. 9.3 illustrates how the JCA architecture is used by commercial products to enable legacy business applications to be accessed from Java applications running on distributed J2EE application servers.



- *JSimpleCalculator*: Java application that provides a new front-end to the SMPLCALC COBOL application.
- *SmpICalcInterfaceHelper*: helper class for building the interface COBOL data structure, can be generated by a commercial product such as RAD.
- *Java to COBOL Marshaller*: class library that implements marshalling of Java data types to/from COBOL data types, likely comes with a commercial J2EE server such as WebSphere Application Server (WAS).

Figure 9.3. Example JCA implementation for accessing a legacy application.

A popular alternative to using the JCA architecture to reengineer and reuse legacy applications is to implement a Service Oriented Architecture (SOA) [38]. When migrating a legacy software system to an SOA, application programs that are candidates for reuse are identified. Typically, candidate applications should be well structured such that the business logic can be isolated, encapsulated, and made into reusable components. These SOA components become capable of communicating without the tight and fragile coupling of traditional binary interfaces because they are wrapped with a platform-neutral interface such as XML and Web services. Once a business or organization has created a collection of reusable components from stable and well tested code, it becomes possible to quickly assemble new applications without having to rewrite and test the

underlying business logic.

When XML is used as envisioned, all data, both of type character and numeric are represented as printable text, completely divorced from any platform-specific representation or encoding. The net effect is that two entities or programs can interact without having to know the data structures that comprise each other's binary interface. Of course, the XML that is exchanged cannot be arbitrary, so industry standards such as XML Schema (XSD) [39], [40] and Web Services Definition Language (WSDL) [41], [42] were developed. XML Schema is used to formally describe XML documents, while WSDL is used to describe services and the operations they support. Operations in Web services are akin to public methods in the object-oriented programming paradigm. A Web service is considered to be WS-I compliant [43], or generally interoperable, if it meets many criteria, not the least of which is using XML documents for the input and output of each operation. There are many criteria defined by WS-I that apply to a Web service definition, but this particular facet, where XML is *the* interoperable interface of choice, sets the stage for a meaningful exercise where the focus is on the activity of making a component from a COBOL program that is reusable from Java using XML in a light-weight, local environment.

In recent history, the ability to parse and generate XML documents has been added to the COBOL language in many implementations including the Micro Focus and IBM COBOL compilers and runtimes [37], [44]. XML parsing in COBOL is accomplished through the use of the XML PARSE statement, which performs an event-

driven parse of an XML document. In a event-driven parse, the initiator registers a handler which the XML parser invokes with each XML construct found in the document. For example, the start and end of an XML element would be reported as two separate events. XML generation in COBOL is accomplished through the use of the XML GENERATE statement, which, given a COBOL data structure and an output buffer, will generate XML that has the same hierarchical organization as the data structure [37, 44]. By default, the XML GENERATE statement will form XML element and attribute names using the name of each member in the COBOL data structure. This can be less than ideal in circumstances where data structure members have cryptic names that don't conform to the spirit of XML where each XML element and attribute is given a name that describes its content. Fortunately, Micro Focus COBOL provides the capability to assign custom XML element and attribute names to each data structure member, which allows for defining an XML Schema that has meaningful element and attribute names [37].

In the exercise which accompanies this section, we are asked to create a language-neutral XML interface to the “legacy” *SMPALC.cbl* application program and invoke it from a Java program which incidentally makes it reusable to other Java programs. To describe an XML interface to the legacy COBOL program so that other programs may consume it, an XML Schema must be created; this can be done with a tool that can generate XML Schema from a COBOL data declaration, or by hand using an XML editor. Once an XML interface has been described using XML Schema, it is necessary to implement XML marshalling layers between the calling Java program and the legacy

COBOL program. In the example exercise, the XML marshalling layer for each program is implemented in the target language itself. So that the Java program can generate and consume XML based on the XML Schema that describes the interface to the COBOL program, we employ the Java Architecture for XML Binding (JAXB) [35]. JAXB facilitates the conversion of Java objects to XML and vice versa. Sun's Java JDK includes a command-line utility *xjc* which generates Java marshalling code from an XML Schema—making it quite easy to write a Java program which consumes and generates XML based on an XML Schema. While generation of XML is nicely handled in COBOL by the XML GENERATE statement, consuming XML involves coding an event handler for the XML PARSE statement. Of course, complete code for both the Java and COBOL XML marshalling layers is included in the solution to the exercise, so if COBOL is a foreign language to you, there's no need for concern. Once the XML marshalling layers are in place, there's one more loose end that needs to be tied up; and that is to figure out how to pass XML documents between the two layers. Since we are in a local scenario, TCP/IP is not an option, therefore a thin Java Native Interface (JNI) layer is needed through which the Java and COBOL marshalling layers can exchange XML; note that the COBOL XML marshalling layer invokes the legacy COBOL application. Fig. 9.4 illustrates the program architecture for the exercise.

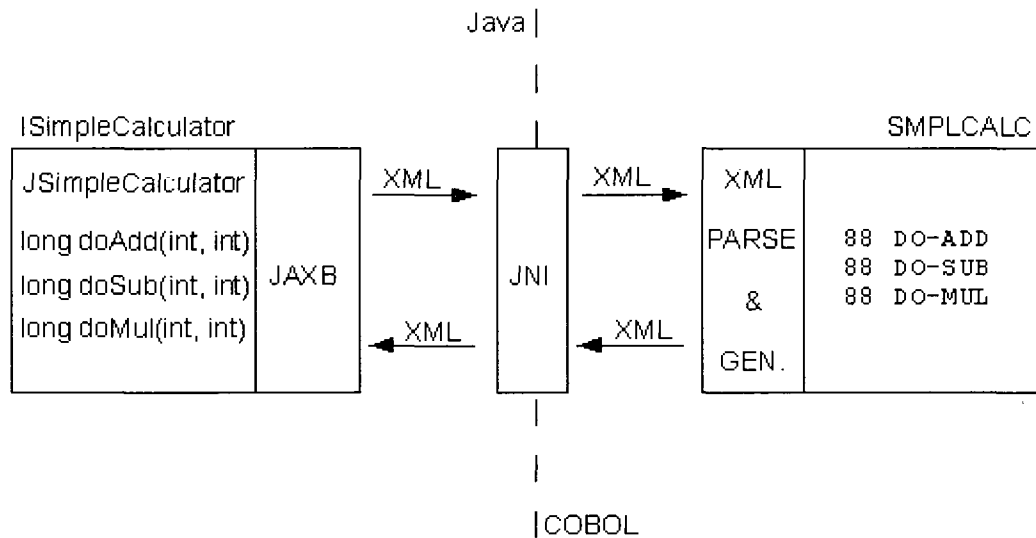


Figure 9.4. Architecture for legacy application reengineering and reuse from Java.

In order to try out the code in this section and complete the exercise that accompanies it, a COBOL compiler and runtime environment are needed. The COBOL programs in this section, and in the solution to the exercise which accompanies it, were written, compiled, and run using a student version of Micro Focus Net Express [37]. At the time of this writing, no reasonably functional open source COBOL compiler was available that could compile, link, and run even the most simple COBOL program given in this section; this may have to do with the fact that COBOL remains a very lucrative enterprise for many businesses, so there is little interest in giving away implementations to the open source community. For example, the COBOL for GCC project has not made significant progress yet on the code generation part of the compiler [36]. When and if an open source COBOL compiler gets off the ground, it will be interesting to see what features of the commercial COBOL compilers are implemented.

9.1 Legacy Software Reengineering and Reuse Exercise

Provide a command-line (or graphical) interactive Java front-end to the legacy COBOL application *SMPLCALC.cbl* by implementing the program architecture illustrated in Fig. 9.4. Before starting the exercise, download and extract the following archive file located here http://reversingproject.info/repository.php?fileID=9_1_1.

Follow these steps to complete the exercise:

- 1) Locate the interface data structure for *SMPLCALC.cbl* in the copybook (source include file) *SMPLCALC.cpy*. There is only one data structure in the copybook.
- 2) Create an XML Schema which represents all of the data in the *SMPLCALC-INTERFACE* COBOL data structure. Instead of writing this by hand, you can use the Micro Focus Net Express CBL2XML wizard [37].
- 3) Write a Java interface *ISimpleCalculator.java* for three computation types supported by *SMPLCALC.cbl* using appropriate method signatures:
 - a) long doAdd(int, int) throws java.lang.ArithmeticException.
 - b) long doSubtract(int, int) throws Java.lang.ArithmeticException
 - c) long doMultiply(int, int) throws Java.lang.ArithmeticException
- 4) Write a Java class *JSimpleCalculator.java* that implements the interface defined in *ISimpleCalculator.java* and provides a user interface for:
 - a) Specifying which computation (add, sub, mul) is desired.
 - b) Specifying the operands to the computation.
 - c) Displaying the result of the computation (can be an error).
- 5) Use the Java command-line utility *xjc*, in combination with the XML Schema created in Step 2, to generate Java to XML marshalling code (JAXB). Update *JSimpleCalculator.java* to call this marshalling code.

- 6) Write a small C/C++ JNI program *Java2CblXmlBridge.cpp* which exports a method “Java2SmplCalc” which:
 - a) Invokes *XML2CALC.cbl* (see Step 7), passing the XML document received from *JSimpleCalculator.java*.
 - b) Returns the XML document generated by *XML2CALC.cbl* (see Step 7) on return from *SMPLCALC.cbl* to *JSimpleCalculator.java*.
- 7) Write a COBOL program *XML2CALC.cbl* which:
 - a) Marshalls XML received from the *Java2CblXmlBridge.cpp*, based on the XML Schema created in Step 2, into *SMPLCALC-INTERFACE*.
 - b) Invokes *SMPLCALC.cbl*, passing *SMPLCALC-INTERFACE* by reference.
 - c) Marshalls *SMPLCALC-INTERFACE* back to XML before returning to *Java2CblXmlBridge.cpp*.
- 8) Compile *XML2CALC.cbl* and link it with the machine/object code for *SMPLCALC.cbl* (*SMPLCALC.obj*).
 - a) To simulate a situation where only partial source code for an application is available, do not recompile *SMPLCALC.cbl*; use the object file (machine code) that comes with this exercise instead.
- 9) Create a DLL that can be loaded and used by *JSimpleCalculator.java* by compiling and linking *Java2CblXmlBridge.cpp* with the object code for *XML2CALC.cbl*.
- 10) Update *JSimpleCalculator.java* to use the *XJC*-generated marshalling code to send/receive XML through the JNI method defined in Step 8 and display the results of the computations performed downstream by *SMPLCALC.cbl*.

9.2 Legacy Software Reengineering and Reuse Exercise Solution

This section gives a solution to the exercise given in Section 9.1. The details of the solution are organized according to the steps of the exercise. Software requirements to build and test the solution include: Sun's Java JDK SE V6, Microsoft Visual C++ Studio Express 2008, and Micro Focus Net Express v5.1 (COBOL).

Most of the source listings in this section are abbreviated, and some of the steps are skipped. The complete source and binaries for the solution can be downloaded from http://reversingproject.info/repository.php?fileID=9_2_1.

1) *Locate the interface data structure for `SMPLCALC.cbl` in the copybook (source include file) `SMPLCALC.cpy`. There is only one data structure in the copybook.*

The interface data structure for `SMPLCALC.cbl` is located in `SMPLCALC.cpy` and is named `SMPLCALC-INTERFACE` (see Table 9.2). COBOL data structures begin with a level 01 declaration and are usually hierarchical but can be elementary.

Table 9.2. Interface data structure `SMPLCALC-INTERFACE` in `SMPLCALC.cpy`.

```
01: * Input/Output data structure
02: 01 SMPLCALC-INTERFACE.
03:     02 SI-OPERAND-1 PIC S9(9) COMP-5.
04:     02 SI-OPERAND-2 PIC S9(9) COMP-5.
05:     02 SI-OPERATION PIC X.
06:         88 DO-ADD VALUE '+'.
07:         88 DO-SUB VALUE '-'.
08:         88 DO-MUL VALUE '*'.
09:     02 SI-RESULT PIC S9(18) COMP-5.
10:     02 SI-RESULT-MESSAGE PIC X(128).
```

- 2) *Create an XML Schema which represents all of the data in the SMPLCALC-INTERFACE COBOL data structure. Instead of writing this by hand, you can use the Micro Focus Net Express CBL2XML wizard [37].*

The CBL2XML wizard in Micro Focus Net Express conveniently generates an XML

Schema from a COBOL data structure. The result of using SMPLCALC.cpy as input to

the CBL2XML wizard is given in Table 9.3.

Table 9.3. XML Schema generated from the COBOL data structure.

```

01: <?xml version="1.0" encoding="UTF-8"?>
02: <schema xmlns="http://www.w3.org/2001/XMLSchema"
03:         elementFormDefault="qualified">
04:   <element name="SMPLCALC-INTERFACE">
05:     <complexType>
06:       <sequence>
07:         <element name="SI-OPERAND-1">
08:           <simpleType>
09:             <restriction base="integer">
10:               <totalDigits value="9" />
11:             </restriction>
12:           </simpleType>
13:         </element>
14:         <element name="SI-OPERAND-2">
15:           <simpleType>
16:             <restriction base="integer">
17:               <totalDigits value="9" />
18:             </restriction>
19:           </simpleType>
20:         </element>
21:         <element name="SI-OPERATION">
22:           <simpleType>
23:             <restriction base="string">
24:               <enumeration value="+" />
25:               <enumeration value="-" />
26:               <enumeration value="*" />
27:             </restriction>
28:           </simpleType>
29:         </element>
30:         <element name="SI-RESULT">
31:           <simpleType>
32:             <restriction base="integer">
33:               <totalDigits value="18" />
34:             </restriction>
35:           </simpleType>
36:         </element>

```

```
27:         <element name="SI-RESULT-MESSAGE">
28:             <simpleType>
29:                 <restriction base="string">
30:                     <maxLength value="128" />
31:                 </restriction>
32:             </simpleType>
33:         </element>
34:     </sequence>
35: </complexType>
36: </element>
37: </schema>
```

- 4)** *Write a Java class `JSimpleCalculator.java` that implements the interface defined in `ISimpleCalculator.java` and provides a user interface for:*
- a) Specifying which computation (add, sub, mul) is desired.*
 - b) Specifying the operands to the computation.*
 - c) Displaying the result of the computation (can be an error).*

There is a great deal of flexibility in this part of the exercise. Some examples of the types of user interfaces that can be implemented include: command-line interactive (console-based), graphical, Java servlet (Web-based). A command-line interactive interface was implemented for the solution. A screen capture of the interface is given Fig. 9.5. Notice that a debugging mode is available to trace the various steps in the process of exchanging XML between the Java and COBOL XML marshalling layers.

```

*****
** Program: Java Front-end to COBOL Calculator      **
** Purpose: Demonstrate reengineering and reuse    **
**           of a COBOL program from Java by       **
**           establishing an XML bridge leveraging  **
**           JAXB, JNI, and COBOL XML support.     **
** Author:  Teodoro Cipresso                       **
**          tcipress@hotmail.com                   **
*****

Select a task from the following menu:

(1) Addition
(2) Subtraction
(3) Multiplication
(4) Toggle Debug ON
(5) Quit Program

Specify selection: 3

Specify integer operand #1: 12

Specify integer operand #2: 12

[***] COBOL multiplication result: 144

```

Figure 9.5. Console-based Java interface to the legacy COBOL program.

- 5) *Use the Java command-line utility `xjc`, in combination with the XML Schema created in Step 2, to generate Java to XML marshalling code (JAXB). Update `JSimpleCalculator.java` to call this marshalling code.*

The `xjc` command-line utility generates two types of artifacts for each global (top level) element in an XML Schema: (1) Java classes that expose getters and setters for the data contained in instances of the XML Schema (XML documents), (2) Java classes that serve as metadata for the JAXB XML marshalling engine. In the solution archive file, the two classes generated by JAXB are: `SmplCalcJaxbFactory.java` (getters and setters) and `SmplCalcJaxbMarshaller.java` (JAXB XML marshalling metadata). Note these are not the default class names generated by `xjc`.

To cleanly integrate the JAXB marshalling with *JSimpleCalculator.java*, *SmplCalcJaxbMarshaller.java*, which encapsulates the interaction with the JAXB, was created. Table 9.4 gives an abbreviated listing of this class.

Table 9.4. Partial listing of *SmplCalcJaxbMarshaller.java* interaction with JAXB.

```
01: private static JAXBContext jaxbContext = null;
02: private static Marshaller marshaller = null;
03: private static Unmarshaller unmarshaller = null;
04:
05: static
06: {
07:     try
08:     {
09:         jaxbContext = JAXBContext.newInstance(SMPLCALCINTERFACE.class);
10:         marshaller = jaxbContext.createMarshaller();
11:         unmarshaller = jaxbContext.createUnmarshaller();
12:     } catch (JAXBException _je) {...}
13: }
14:
15: public static String serializeXML(SMPLCALCINTERFACE request)
16: {
17:     ByteArrayOutputStream xmlBytes = new ByteArrayOutputStream();
18:     try
19:     {
20:         marshaller.marshal(request, xmlBytes);
21:     } catch (JAXBException _je) {...}
22:     String xmlDoc = new String(xmlBytes.toByteArray());
23:     return xmlDoc;
24: }
25:
26: public static SMPLCALCINTERFACE loadXML(String xmlDoc)
27: {
28:     SMPLCALCINTERFACE response = null;
29:     ByteArrayInputStream xmlBytes = new
ByteArrayInputStream(xmlDoc.getBytes());
30:     try
31:     {
32:         response = (SMPLCALCINTERFACE)unmarshaller.unmarshal(xmlBytes);
33:     } catch (JAXBException _je) {...}
34:     return response;
35: }
```

Next we need to update the add, subtract, and multiply methods in *JSimpleCalculator.java* to use *SmplCalcJaxbMarshaller.java* to generate and consume

XML in preparation to use the JNI XML bridge to the legacy COBOL application. Table 9.5 contains an abbreviated listing of the updated to *JsimpleCalculator.java*. Note that the call to method *smplCalcXmlInterface* is commented out. This is a call to the JNI XML bridge which will be implemented in a later step.

Table 9.5. Updates to *JSimpleCalculator.java* in support of JAXB marshalling.

```

01: public long doAdd(int _1stOp, int _2ndOp)
02: {
03:     SMPLCALCINTERFACE addResult = invokeXmlInterface("+", _1stOp,
04:         _2ndOp);
05:     return addResult.getSIRESULT().longValue();
06: }
07: public SMPLCALCINTERFACE invokeXmlInterface(String calcType, int
08:     _1stOp, int _2ndOp)
09: {
10:     SMPLCALCINTERFACE inputData = new SmplCalcJaxbFactory().
11:     createSMPLCALCINTERFACE();
12:     inputData.setSIOPERATION(calcType);
13:     inputData.setSIOPERAND1(BigInteger.valueOf(_1stOp));
14:     inputData.setSIOPERAND2(BigInteger.valueOf(_2ndOp));
15:     inputData.setSIRESULTMESSAGE("");
16:     inputData.setSIRESULT(BigInteger.valueOf(0));
17:     String inputXml = SmplCalcJaxbMarshaller.serializeXML(inputData);
18:     // TODO JNI: String outputXml = smplCalcXmlInterface(inputXml);
19:     SMPLCALCINTERFACE outputData = SmplCalcJaxbMarshaller.
20:     loadXML(outputXml);
21:     return outputData;
22: }

```

- 6) Write a small C/C++ JNI program *Java2CblXmlBridge.cpp* which exports a method "Java2SmplCalc" which:
- a) Invokes *XML2CALC.cbl* (see Step 7), passing the XML document received from *JSimpleCalculator.java*.
 - b) Returns the XML document generated by *XML2CALC.cbl* (see Step 7) on return from *SMPLCALC.cbl* to *JSimpleCalculator.java*

Sun's Java SDK includes the command-line utility *javah* that generates appropriate C/C++ header files for a *native* method declaration in a Java class. The

generated header file will contain a function prototype that reflects the fully qualified name and signature of the method. Using the function prototype, it is the responsibility of the programmer to write a C/C++ method that conforms to it and interacts properly with the JVM. Please note that garbage collection does not apply to any memory allocated by the native code, so be sure to free it.

To generate the JNI header file, we must first declare a native method in *JsimpleCalculator.java* that we wish to implement in C/C++. In addition, we must also indicate the name of the DLL Java will need to load in order to call it. Table 9.6 contains the needed additions to *JsimpleCalculator.java* to declare the native method. Note that on the *System.loadLibrary* call, the file extension of the DLL file is not specified.

Table 9.6. Example native method declaration for the JNI XML bridge.

```
01: public class JSimpleCalculator implements ISimpleCalculator
02: {
03:     native String smplCalcXmlInterface(String xmldoc);
04:     static
05:     {
06:         System.loadLibrary("Java2CblXmlBridge");
07:     }
08:     ...
09: }
```

When using the *javah* command-line utility, keep in mind that it operates on *.class files instead of *.java files; this is because the Java reflection APIs are used to get the qualified name and signature of the native method declaration instead of having to parse the source file. To generate a C/C++ header file from the *JSimpleCalculator.class* file, issue the command “*javah -jni info.reversingproject.jsimplecalculator.JSimpleCalculator*.” Table 9.7 gives the source

for the JNI program *Java2CblXmlBridge.cpp*, which implements the JNI method described in the generated header file.

Table 9.7. Example implementation of the Java to COBOL JNI XML bridge.

```

01: #include "package_JSimpleCalculator.h"
02: #include "cobcall.h"
/*
 * Class:      info_reversingproject_jsimplecalculator_JSimpleCalculator
 * Method:     smplCalcXmlInterface
 * Signature:  (Ljava/lang/String;)Ljava/lang/String;
 */
03: jstring JNICALL
Java_info_reversingproject_jsimplecalculator_JSimpleCalculator_smplCalc
XmlInterface (JNIEnv *env, jobject parent_objct, jstring xml_doc)
04: {
05:     // Get input XML document passed from Java
06:     jboolean iscopy;
07:     jstring output_xml;
08:     char *xml_buffer = NULL;
09:     char *xml_buffer_ptr = NULL;
10:     const char *xml_input = (*env)->GetStringUTFChars(env, xml_doc,
&iscopy);
11:     int xml_len = strlen(xml_input);
12:     // Allocate XML I/O buffer and copy input XML
13:     xml_buffer = (char*)malloc(32767);
14:     memset(xml_buffer, 0x00, 32767); // initialize
15:     memcpy(xml_buffer, xml_input, xml_len);
16:     // Free JNI memory used for MBCS to SBCS conversion
17:     (*env)->ReleaseStringUTFChars(env, xml_doc, &iscopy);
18:     // call COBOL to XML marshalling layer, passing XML I/O buffer
19:     cobinit(); // Initialize Micro Focus COBOL runtime
20:     XML2CALC(&xml_len, xml_buffer); // Call COBOL
21:     // Null terminate XML returned from COBOL
22:     xml_buffer_ptr = xml_buffer;
23:     xml_buffer_ptr += xml_len;
24:     *(xml_buffer_ptr) = 0x00;
25:     // Allocate UTF version of XML to return to Java
26:     output_xml = (*env)->NewStringUTF(env, xml_buffer);
27:     // Free XML I/O buffer
28:     free(xml_buffer);
29:     // Return XML generated by COBOL as Java String
30:     return output_xml;
31: }

```

- 7) Write a COBOL program *XML2CALC.cbl* which:
- a) Marshalls XML received from the *Java2CblXmlBridge.cpp*, based on the XML Schema created in Step 2, into *SMPLCALC-INTERFACE*.
 - b) Invokes *SMPLCALC.cbl*, passing *SMPLCALC-INTERFACE* by reference.
 - c) Marshalls *SMPLCALC-INTERFACE* back into XML document before returning to *Java2CblXmlBridge.cpp*.

Using the recently added XML support in COBOL [37, 44], parsing and generation of XML is fairly straight-forward. Two statements in the COBOL language, XML PARSE and XML GENERATE, are used to implement the program *XML2CALC.cbl*. Note that the XML GENERATE statement only allows assignment of non-default XML element names to data structure members when reading or writing from an XML file. Since we are working with XML in a stream, the XML Schema defined in the solution to Step 2 uses the default XML element names generated by the Micro Focus Net Express CBL2XML wizard. Table 9.8 gives the source code for *XML2CALC.cbl*, the XML layer to the legacy COBOL application.

Table 9.8. Implementation of a COBOL XML layer to the legacy application.

```

01: $set preprocess(prexml) o(foo.pp) warn endp
02: *****
03: ** Wrapper program that provides an XML interface to SMPLCALC **
04: *****
05: IDENTIFICATION DIVISION.
06: PROGRAM-ID. 'XML2CALC'.
07: DATA DIVISION.
08: WORKING-STORAGE SECTION.
09: * Input/Output data structure
10: 01 SMPLCALC-INTERFACE.
11: 02 SI-OPERAND-1 PIC S9(9) COMP-5.
12: 02 SI-OPERAND-2 PIC S9(9) COMP-5.
13: 02 SI-OPERATION PIC X.
14: 88 DO-ADD VALUE '+'.
15: 88 DO-SUB VALUE '-'.
16: 88 DO-MUL VALUE '*'.
17: 02 SI-RESULT PIC S9(18) COMP-5.
18: 02 SI-RESULT-MESSAGE PIC X(128).
19: * XML parsing state
20: 01 XML-PARSE-STATE.
21: 02 CURR-ELE-NAME PIC X(256).
22: 02 CURR-ELE-CONT PIC X(256).
23: LINKAGE SECTION.
24: 01 XML-DOC-LEN PIC S9(9) COMP-5.
25: 01 XML-DOC-TXT PIC X(32767).
26: PROCEDURE DIVISION USING XML-DOC-LEN XML-DOC-TXT.
27: MAINLINE SECTION.
28: * Parse XML into SMPLCALC-INTERFACE
29: XML PARSE XML-DOC-TXT(1:XML-DOC-LEN)
30: PROCESSING PROCEDURE XML-HANDLER
31: END-XML
32: * Invoke legacy COBOL application SMPLCALC
33: CALL 'SMPLCALC' USING SMPLCALC-INTERFACE
34: * Generate XML from SMPLCALC-INTERFACE
35: XML GENERATE XML-DOC-TXT FROM SMPLCALC-INTERFACE
36: COUNT IN XML-DOC-LEN
37: END-XML
38: * Return to client program
39: GOBACK
40: .
41: * +-----+
42: * | XML event handler for marshalling XML into COBOL data |
43: * +-----+
44: XML-HANDLER.
45: EVALUATE XML-EVENT
46: WHEN 'START-OF-ELEMENT'
47: MOVE XML-TEXT TO CURR-ELE-NAME
48: WHEN 'CONTENT-CHARACTERS'
49: EVALUATE CURR-ELE-NAME

```

```

50:          WHEN 'SI-OPERAND-1'
51:            MOVE FUNCTION NUMVAL(XML-TEXT) TO SI-OPERAND-1
52:          WHEN 'SI-OPERAND-2'
53:            MOVE FUNCTION NUMVAL(XML-TEXT) TO SI-OPERAND-2
54:          WHEN 'SI-OPERATION'
55:            MOVE XML-TEXT TO SI-OPERATION
56:          END-EVALUATE
57:        WHEN 'END-OF-ELEMENT'
58:          INITIALIZE CURR-ELE-NAME
59:        END-EVALUATE
60:      .
61:    END PROGRAM 'XML2CALC'.

```

10) Update *JSimpleCalculator.java* to use the *XJC*-generated marshalling code to send/receive XML through the JNI method defined in Step 8 and display the results of the computations performed downstream by *SMPLCALC.cbl*.

To begin using the JNI XML bridge, create or uncomment a line in your code that corresponds to the bolded line in Table 9.5. Essentially, code a call to method *Java2CblXmlBridge.smplCalcXmlInterface(inputXmlDoc)*, passing the JAXB generated XML document, to invoke the legacy COBOL application *SMPLCALC.cbl* through JNI and the XML layers. Table 9.9 lists the results of running the complete solution code for the exercise with debug tracing turned on.

Table 9.9. Example run of the solution code with debug statements turned on.

```

01: *****
02: ** Program: Java Front-end to COBOL Calculator **
03: ** Purpose: Demonstrate reengineering and reuse **
04: ** of a COBOL program from Java by **
05: ** establishing an XML bridge leveraging **
06: ** JAXB, JNI, and COBOL XML support. **
07: ** Author: Teodoro Cipresso **
08: ** tcipress@hotmail.com **
09: *****
10:
11: Select a task from the following menu:
12:

```

```

13: (1) Addition
14: (2) Subtraction
15: (3) Multiplication
16: (4) Toggle Debug OFF
17: (5) Quit Program
18:
19: Specify selection: 3
20:
21: Specify integer operand #1: 16
22:
23: Specify integer operand #2: 32
24:
25: [D] JSimpleCalculator.doMultiply(16, 32)
26:
27: [D] JSimpleCalculator.invokeXmlInterface(*, 16, 32)
28:
29: [D] SmplCalcJaxbMarshaller.serializeXML()
30:
31: [D] SmplCalcJaxbMarshaller.serializeXML().xmlDoc[<?xml
version="1.0" encoding="UTF-8" standalone="yes"?><SMPLCALC-
INTERFACE><SI-OPERAND-1>16</SI-OPERAND-1><SI-OPERAND-2>32</SI-OPERAND-
2><SI-OPERATION>*</SI-OPERATION><SI-RESULT>0</SI-RESULT><SI-RESULT-
MESSAGE></SI-RESULT-MESSAGE></SMPLCALC-INTERFACE>]
32:
33: [D] JSimpleCalculator.invokeXmlInterface(): Before call to
Java2CblXmlBridge
34:
35: [D] JSimpleCalculator.invokeXmlInterface(): After call to
Java2CblXmlBridge
36:
37: [D] SmplCalcJaxbMarshaller.loadXML().xmlDoc[<SMPLCALC-
INTERFACE><SI-OPERAND-1>16</SI-OPERAND-1><SI-OPERAND-2>32</SI-OPERAND-
2><SI-OPERATION>*</SI-OPERATION><SI-RESULT>512</SI-RESULT><SI-RESULT-
MESSAGE>Completed successfully</SI-RESULT-MESSAGE></SMPLCALC-
INTERFACE>]
38:
39: ***] COBOL multiplication result: 512

```

10 Identifying, Monitoring, and Reporting Malware

Malware describes a category of software that for one reason or another does not fit the description of a program that always operates in a way that benefits the user [5]. Of course, those of us who have ever used software might contend that this definition of malware will cause programs that we use every day to be categorized as malware. For example, the word processor used to write this paragraph has crashed more than once during the writing of this paper, and, in that regard, it's not acting in a way that benefits the user. To tighten the definition of malware, let's qualify it a bit: the malicious or annoying behaviors of malware are *intentional*, not the result of one or more bugs. There are currently five types of malware that affect computer systems [5] [21]:

- *Viruses*: a virus is malware that requires some deliberate action to help it spread. For example, a user downloading and installing an infected program that in turn infects emails sent by the user.
- *Worms*: a worm is similar to a virus but can spread by itself over computer networks. Worms have superseded viruses as the popular choice of hackers.
- *Trojan horses*: a Trojan horse is software that has hidden and unadvertised functionality that occurs during normal use.
- *Backdoor*: a backdoor is a vulnerability purposely embedded in software that allows an attacker to connect to the users machine with malicious intent.
- *Rabbit*: a rabbit is a program that exhausts system resources. Types of resources that can be exhausted include memory, disk space, CPU time.

To experiment with most of the types of malware listed here is dangerous. Therefore, if one decides to try one's hand at analyzing real-life malware, using the machine code and bytecode reversing techniques demonstrated in this paper, one should do so in a carefully prepared environment. One should not install *any* malware on a computer that must remain in operating condition. Worms and backdoors can be especially dangerous because they can propagate to other systems on computer networks. Be aware that using virtualization tools such as VMware to create secondary operating system images on which to install malware can still result in the infection of the primary operating system, especially if the VMware-hosted image has connectivity enabled.

The goal of this section is to help you become familiar with using software tools to identify, monitor, report, and securely delete software that you suspect to be malicious. Since it's not practical to ask that you install a virus, worm, backdoor, or rabbit on your machine, we are left with the possibility of a *guaranteed benign* software Trojan. It's important to note here that malware usually isn't of just one type; for example, 3 of the top 10 malicious codes families reported in 2008 were Trojans with a backdoor component [45]. It turns that focusing on software Trojans is appropriate because as Symantec's 2009 Global Internet Security Threat Report [45] states, "Trojans made up 68 percent of the volume of the top 50 malicious code samples reported in 2008", and "Five of the top 10 staged downloaders in 2008 were Trojans."

For the vast majority of us, the story of the Trojan horse from antiquity is quite familiar. Essentially, the Greeks, in a 10-year siege against the city of Troy, devised a

brilliant plan of putting 40 of their best soldiers into the body of a large wooden horse while the rest of the army sailed away out of sight. The Trojans, assuming that the Greeks had given up, pulled the horse into their city as a trophy of their victory. As night fell over the city of Troy, the Greek army sailed back to shore. Meanwhile, the soldiers in the Trojan horse silenced some guards and opened the gates—allowing the Greek army to flood in and take the city by surprise.

So what does all this have to do with software? Not too surprising, a Trojan software program is one that is not entirely what it seems. For example, imagine a program is offered for free on the Internet that claims to be able to convert audio files between different formats. The program fits the needs of many, and is definitely the right price, so it has a large install base. What users of the program are not told is that while the program is performing its advertised functions, it will perform other annoying or malicious tasks in the background such as: scanning the system for sensitive information and uploading it to a rogue site, affecting the stability and performance of the system by doing repeated expensive operations.

In 1996, Mark Russinovich founded a company called “Winternals Software” where he was the chief software architect on a comprehensive suite of tools for diagnosing, debugging, and repairing Windows® systems and applications [46]. Mark's company has since been purchased by Microsoft and his suite of tools have been rebranded “Windows Sysinternals” and are offered for free on Microsoft Technet. An example of one of the more powerful tools in the Sysinternals suite is the Process

Monitor. The Process Monitor can capture detailed information about any running process in a Windows® system including: filesystem, registry, and network activity. Just the Process Monitor alone is helpful in analyzing the behavior of an application when making the determination of whether or not it is malicious. As an aside, Mark's story is an interesting one because he is recognized as a true expert on the internals of Windows® even though he did not participate in its development—a true testament to what can be learned about software through reverse engineering. At the time of this writing, the Sysinternals suite contained 66 different utilities, but we'll focus on the most useful one in this context of analyzing the behavior of malware: Process Monitor. In the exercise that accompanies this section, it is recommended that you use Process Monitor to complete it. If you have the opportunity to experiment with other tools in the Sysinternals suite, you are encouraged to do so. The following description of Process Monitor is given on the Windows Sysinternals web site [46]:

“Process Monitor is an advanced monitoring tool for Windows® that shows real-time file system, Registry and process/thread activity. It combines the features of two legacy Sysinternals utilities, Filemon and Regmon, and adds an extensive list of enhancements including rich and non-destructive filtering, comprehensive event properties such session IDs and user names, reliable process information, full thread stacks with integrated symbol support for each operation, simultaneous logging to a file, and much more. Its uniquely powerful features will make Process Monitor a core utility in your system troubleshooting and malware hunting toolkit.”

Fig. 10.1 contains a capture of a Process Monitor session where the filesystem activity of the Password Vault application is recorded. When using Process Monitor, you can selectively monitor registry, filesystem, network, and thread activity.

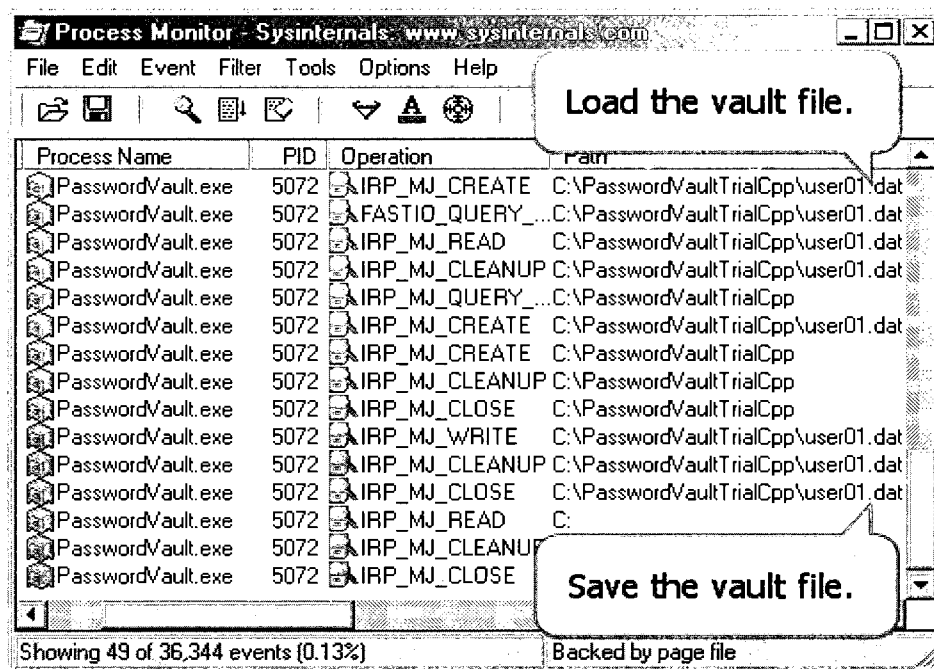


Figure 10.1. Process Monitor session for the Password Vault application.

Most of the malicious operations carried out by Trojans can be detected using Process Monitor, including those that contain Backdoors. Of course, Process Monitor itself doesn't identify malware, it simply reports what a process is doing. With a little bit of ingenuity, one can identify activities that don't seem to fit with the advertised functionality of a program. For example, a program that accesses registry keys, files, or network locations that are unrelated to it, is probably malicious. It's common practice these days for users to download free software from the Internet, and because we've been convinced that open-source software, which is sometimes confused with free software, should have the fewest number of vulnerabilities, we do it without much afterthought. Incidentally, the data on the number of vulnerabilities found in popular Internet browsers

does not support this belief. [45] reports that “Mozilla browsers were affected by 99 new vulnerabilities in 2008, more than any other browser; there were 47 new vulnerabilities identified in Internet Explorer, 40 in Apple Safari, 35 in Opera™, and 11 in Google® Chrome.” It seems counter-intuitive that an open-source browser would have twice as many security holes than a closed-source browser like Internet Explorer. Mozilla is not malware, but it's interesting to note that in the case of software, open-source doesn't guarantee security. Becoming familiar with the Windows® Sysinternals suite can help you evaluate whether the software on your Windows® machine is acting in your best interest.

If you suspect a particular program to be malware, it can be submitted online to a service called ThreatExpert [47]. ThreatExpert is a Web-based tool that supports submission of software executables that are to be evaluated against an on-line malware database. The tool analyzes the instruction sequences in submitted executables and attempts to match them against those of known malware. Matching against existing malware is just one part of ThreatExpert's automated engine; the service actually tries to execute suspected malware in an isolated environment in order to perform heuristic analysis of its actions. An example of a report generated by ThreatExpert for a particularly dangerous piece of malware is shown in Fig. 10.2. The figure contains only the top-level summary of the report whereas the full report contains much more detail, such as filesystem, memory, registry, network and other activity. Note that all of the malicious behaviors of the submitted executable could have been learned by



Submission Summary:

Submission details:

- ▶ Submission received: 2 May 2009, 13:53:25
- ▶ Processing time: 6 min 33 sec
- ▶ Submitted sample:
 - └ File MD5: 0xD5D9730AF8DE7006C9940791E96B20CE
 - └ File SHA-1: 0xC4AD816CC3AD6206735E24903DC58729AAB6B388
 - └ Filesize: 406,771 bytes
 - └ Alias:
 - └ Virus.Win32.Parite.b ▶ [Kaspersky Lab]
 - └ Virus.Win32.Parite ▶ [Ikarus]

Summary of the findings:

What's been found	Severity Level
A network-aware worm that uses known exploit(s) in order to replicate across vulnerable networks.	■■■■■■■■■■
MS04-011: LSASS Overflow exploit - replication across TCP 445 (common for Sasser, Bobax, Kibuv, Korgo, Gaobot, Spybot, Randex, other IRC Bots).	■■■■■■■■■■
Replication across networks by exploiting weakly restricted shares (common for Randex family of worms).	■■■■■■■■■■
Communication with a remote IRC server.	■
Downloads/requests other files from Internet.	■
Creates a startup registry entry.	■■
There were some system executable files modified, which might indicate the presence of a PE-file infector.	■■■■■
Contains characteristics of an identified security risk.	■■■■■■■■■■

Figure 10.2. Example ThreatExpert report summary for submitted malware.

monitoring it using Process Monitor, though it would have taken much more time.

To facilitate the exercise which accompanies this section, a benign Java software

Trojan named “Alarm Clock” was written. The Alarm Clock program is a multi-threaded, console-based application that allows you to interact with it while it continually checks whether or not to sound the alarm. Obviously, the Alarm Clock program does a bit more than its advertised function, and the goal of the exercise is to help build familiarity with the Windows Sysinternals tool suite through attempting to figure out what the additional actions taken by the program are. Keep in mind that malware will not necessarily accomplish its goals as quickly possible, it may spread out or pace malicious activity in order to use fewer system resources—helping it stay under the radar of the user. The user interface of the Alarm Clock application is shown in Fig. 10.3.

```
+-----+
|           Alarm Clock V1.0           |
+-----+
(1) Display the current date and time.
(2) Display the alarm date and time.
(3) Set the alarm date and time.
(4) Quit.

>> Type an option number and press Enter: 1

[INFO] The current time is (05/02/09 13:49:48).

+-----+
|           Alarm Clock V1.0           |
+-----+
(1) Display the current date and time.
(2) Display the alarm date and time.
(3) Set the alarm date and time.
(4) Quit.

>> Type an option number and press Enter: 3

>> Specify the alarm date and time...(mm/dd/yy HH:MM:SS).
>> The current date and time is (05/02/09 13:49:53).
>> Type the alarm date and time to set ==> 05/03/09 08:00:00

[INFO] Alarm set is successful.
```

Figure 10.3. Console-based UI for the Alarm Clock example software Trojan.

10.1 Malware Identification and Monitoring Exercise

Using the Windows Sysinternals suite of diagnostic tools, identify the behaviors of the Alarm Clock application that make it a software Trojan. Note any filesystem, memory, registry, or other activity that is unrelated to the program's advertised functionality. The Alarm Clock application is available at the following location:

➤ *Alarm Clock Java Application Windows® installer:*

http://reversingproject.info/repository.php?fileID=10_1_1

Note that even though the Alarm Clock application is written in Java, the bytecode has been aggressively obfuscated to discourage the use of decompilation as a strategy for learning the application's behavior.

10.2 Malware Identification and Monitoring Exercise Solution

The Alarm Clock application is a benign software Trojan that in addition to being a rudimentary alarm clock, collects information about the Windows® installation, and randomly scans for computers on the Internet or Intranet that will respond to an ICMP ping. The application logs all of the information it gathers into several files in a directory off of the root filesystem, or off of the current directory (if the root filesystem is not writeable). The specific information gathered by the application is as follows:

- Registry data on the Windows® installation including the license key.
- Registry data on the currently installed programs.
- The locations of Microsoft Office, OpenOffice, PDF, and text documents in the

“Documents and Settings” folder.

- IP addresses of random Internet/Intranet hosts that respond to an ICMP ping.

Conclusion

Unless something is done to include a required amount of reverse engineering instruction in computer science and software engineering programs of study, new engineers will remain ill-equipped to work with legacy software systems as well as be unable to ensure that software is secure and safe to deploy. Most large companies have existing software systems that have been the underpinning of their business for years. It's highly difficult, not to mention cost-prohibitive, to rip and replace mission-critical software systems in response to the emergence of a new technology. As a result, organizations 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. Students and practicing engineers need reverse engineering skills to be able to help organizations, both large and small, understand their current technology stack and recommend an integration strategy for new technologies. Software security issues, such as how the latest virus or worm infects computer systems, also require extensive reverse engineering knowledge.

Since students and engineers need to learn reverse engineering, instructors need to be able to teach it to them. At the present time, even experienced computer science and software engineering instructors may not have enough knowledge of reverse engineering to teach a course on it. Compounding the problem is the fact that materials for teaching a course on reverse engineering may be difficult to find in a format that is compatible with

classroom delivery. Several books exist on reverse engineering that cater to industry professionals or those interested in self-study. However, in a university setting, instructors engage students in ordered learning through exercises, quizzes, and exams. Since SRE is not a standard part of the computer science curriculum, instructors will be mostly on their own to create a course that they feel gives an adequate education on the subject. Since the uses of software reverse engineering have been well documented in the literature, it is certainly feasible to provide education on the topic, though coming up with good exercises is challenging. The importance of making this education available was emphasized by El-Ramly at the 28th International Conference on Software Engineering when he stated “Reengineering skills are survival skills for those who have to carry out software renovation and modernization projects” [48].

The integration of reverse engineering techniques as part of learning in traditional computer science courses has been tried at the University of Missouri-Rolla [3]. When students were polled, 77% indicated that applying reverse engineering techniques to their normal programming assignments reinforced concepts taught during lectures [3]. Furthermore, 82% of students wanted reverse engineering to be blended in future courses, especially those that dealt with design [3]. Given these promising trials, universities should continue to work toward establishing standard content for software reverse engineering and software maintenance courses.

References

- [1] H. A. Müller, J. H. Jahnke, D. B. Smith, M. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proc. Conf. Future of Software Engineering*, Limerick, Ireland, 2000, pp. 47-60.
- [2] G. Canfora and M. Di Penta, "New Frontiers of Reverse Engineering," in *Proc. Future of Software Engineering*, Minneapolis, MN, 2007, pp. 326-341.
- [3] M. R. Ali, "Why teach reverse engineering?" *ACM SIGSOFT SEN*, v.30, n.4, pp.1-4, Jul 2005.
- [4] A. V. Deursen, J. Favre, R. Koschke, and J. Rilling, "Experiences in Teaching Software Evolution and Program Comprehension," in *Proc. 11th IEEE Int. Workshop on Program Comprehension*, Washington, DC, 2003, pp. 2834-284.
- [5] E. Eliam, *Secrets of Reverse Engineering*, Indianapolis, IN: Wiley, 2005.
- [6] L. Cunningham. (2008, Jul 9). *COBOL Reborn* [Online]. Available: <http://it.toolbox.com/blogs/oracle-guide/cobol-reborn-25896>
- [7] B. W. Weide, W. D. Heym, J. E. Hollingsworth, "Reverse engineering of legacy code exposed," in *Proc. 17th Int. Conf. Software Engineering*, Seattle, Washington, WA, 1995, pp. 327-331.
- [8] Wikipedia contributors. (2008, Sept 9). *Compiler* [Online]. Available: <http://en.wikipedia.org/w/index.php?title=Compiler&oldid=237244781>
- [9] B. Gough, *An introduction to GCC for the GNU Compilers gcc and g++*, Bristol, United Kingdom: Network Theory Limited, 2005.
- [10] K. Irvine, *Assembly Language: For Intel-Based Computers*, Upper Saddle River, NJ: Prentice Hall, 2007.
- [11] Boomerang Decompiler Project. (2006), *Boomerang: a general, open source, retargetable decompiler of machine code programs* (Version 0.3.2) [Online]. Available: <http://boomerang.sourceforge.net>
- [12] Backer Street Software. (2007). *REC: Reverse Engineering Compiler* (Version 2.1) [Online]. Available: <http://www.backerstreet.com/rec/rec.htm>
- [13] O. Yuschuk. (2000). *OllyDbg: 32-bit assembler level analysing debugger for Microsoft Windows®* (Version 1.1) [Online]. Available: <http://www.ollydbg.de>
- [14] Wikipedia contributors. (2008, Oct 2008). *Machine code* [Online]. Available: http://en.wikipedia.org/w/index.php?title=Machine_code&oldid=246690032
- [15] P. Hagggar. (2001, Jul 1). *Java bytecode: Understanding bytecode makes you a better programmer* [Online]. Available: http://www.ibm.com/developerworks/ibm/library/it-hagggar_bytecode

- [16] P. Kouznetsov. (2001). *Jad: 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* (Version 1.5.8g) [Online]. Available: <http://www.kpdus.com/jad.html>
- [17] Wei Dai. (2008). *Crypto++® Library, Crypto++ Library is a free C++ class library of cryptographic schemes* (Version 5.5.2) [Online]. Available: <http://www.cryptopp.com>
- [18] G. M. Weinberg, *The Psychology of Computer Programming*, New York, New York: Dorset House Publishing, 1998.
- [19] A. Kalinovsky, *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*, Indianapolis, IN: Sam's Publishing, 2004.
- [20] A. Sinkov, *Elementary Cryptanalysis: A Mathematical Approach*. Washington, DC: The Mathematical Association of America, 1980.
- [21] M. Stamp, *Information Security: Principles and Practice*, Hoboken, NJ: John Wiley & Sons, 2006.
- [22] Wikipedia contributors. (2009, Feb 9). *ROT13* [Online]. Available: <http://en.wikipedia.org/w/index.php?title=ROT13&oldid=269492700>
- [23] B. Baier. (2006). *COBF: the Freeware C/C++ Sourcecode Obfuscator* (Version 1.06) [Online]. Available: <http://home.arcor.de/bernhard.baier/cobf>
- [24] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308-320, July 1976. Available: <http://www.literateprogramming.com/mccabe.pdf>
- [25] Wikipedia contributors. (2008, Sept 26). *Levenshtein distance* [Online]. Available: http://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=273450805
- [26] Zelix Pty Ltd. (2009). *Zelix Klassmaster: Java Bytecode Obfuscator* (Version 5.2) [Online]. Available: <http://www.zelix.com/klassmaster/features.html>
- [27] The University of Arizona, Department of Computer Science. (2004). *SandMark: A Tool for the Study of Software Protection Algorithms* (Version 3.4) [Online]. Available: <http://sandmark.cs.arizona.edu>
- [28] Retrologic Systems. (2007). *RetroGuard for Java Obfuscation* (Version 2.3.1) [Online]. Available: <http://www.retrologic.com/retroguard-main.html>
- [29] E. Lafortune. (2008). *ProGuard v4.3: a Free Java bytecode Shrinker, Optimizer, Obfuscator, and Preverifier* (Version 4.3) [Online]. Available: <http://proguard.sourceforge.net>
- [30] A. G. Shvets. (1999). *CafeBabe: Graphical Classfile Disassembler, Editor, Stripper, Migrator, Compactor and Obfuscator* (Version 1.2.7.a) [Online]. Available: <http://www.geocities.com/CapeCanaveral/Hall/2334/programs.html>

- [31] M. R. Batchelder, "Java Bytecode Obfuscation", M.S. Thesis, Dept. Comp Sci., McGill Univ., Montreal, Canada, 2007. Available: http://digitoollibrary.mcgill.ca:1801/webclient/StreamGate?folder_id=0&dvs=1236657408333~988
- [32] H. M. Sneed, "Encapsulation of legacy software: A technique for reusing legacy software components", in *Ann. Software Engineering*, v.9, n.4, pp.293-313, 2000.
- [33] IBM, (2008). *IBM® Rational® Application Developer for WebSphere® Software* (Version 7.5.1) [Online]. Available: <http://www-01.ibm.com/software/awdtools/developer/application>
- [34] Sun Microsystems. (2005, May 11). *J2EE Connector Architecture* [Online]. Available: <http://java.sun.com/j2ee/connector>
- [35] Wikipedia contributors. (2009, Mar 24). *Java Architecture for XML Binding* [Online]. Available: http://en.wikipedia.org/w/index.php?title=Java_Architecture_for_XML_Binding&oldid=279402856
- [36] Free Software Foundation. (2000). *COBOL For GCC: a project to produce a free COBOL compiler compliant with the COBOL 85 Standard, integrated into the GNU Compiler Collection (GCC)* (Version 0.1.2) [Online]. Available: <http://cobolforgcc.sourceforge.net>
- [37] Micro Focus Ltd (2008). Net Express Personal Edition: a complete environment for quickly building and modernizing COBOL enterprise components and business applications (Version 5.1) [Online]. Available: <http://www.microfocus.com/Resources/Communities/Academic>
- [38] World Wide Web Consortium contributors. (2004, Feb 11). *Web Services Architecture* [Online]. Available: <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211>
- [39] World Wide Web Consortium contributors. (2004, Oct 28). *XML Schema Part 1: Structures* (2nd ed.) [Online]. Available: <http://www.w3.org/TR/xmlschema-1>
- [40] World Wide Web Consortium contributors. (2004, Oct 28). *XML Schema Part 2: Datatypes* (2nd ed.) [Online]. Available: <http://www.w3.org/TR/xmlschema-2>
- [41] World Wide Web Consortium contributors. (2004, Jun 26). *Web Services Description Language (WSDL) Part 1: Core Language* (Version 2.0) [Online]. Available: <http://www.w3.org/TR/wsdl20>
- [42] World Wide Web Consortium contributors. (2004, Jun 26). *Web Services Description Language (WSDL) Part 2: Adjuncts* (Version 2.0) [Online]. Available: <http://www.w3.org/TR/wsdl20-adjuncts>
- [43] Web Services Interoperability Organization. (2007, Oct 24). *Basic Profile* (Version 1.2) [Online]. Available: [http://www.ws-i.org/Profiles/BasicProfile-1_2\(WGAD\).html](http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html)

- [44] IBM. (2007). *Enterprise COBOL for z/OS: Language Reference V4R1*. (1st ed.) [Online]. Available: <http://publibfp.boulder.ibm.com/epubs/pdf/igy3lr40.pdf>
- [45] Symantec Corp. (2009 Apr). *Symantec Global Internet Security Threat Report* (1st ed.) [Online]. *Volume 14(1)*. Available: http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiv_04-2009.en-us.pdf
- [46] Microsoft TechNet. (2009, May 7). *Windows Sysinternals: utilities to help manage, troubleshoot and diagnose Windows systems and applications*. [Online]. Available: <http://technet.microsoft.com/en-us/sysinternals/default.aspx>
- [47] ThreatExpert Ltd. (2009) *ThreatExpert: ThreatExpert is an advanced automated threat analysis system designed to analyze and report the behavior of computer viruses, worms, trojans, adware, spyware, and other security related risks in a fully automated mode*. [Online]. Available: <http://www.threatexpert.com>
- [48] M. El-Ramly, "Experience in teaching a software reengineering course," in *Proc. 28th Int. Conf. on Software Engineering*. Shanghai, China, 2006, pp. 699-702.