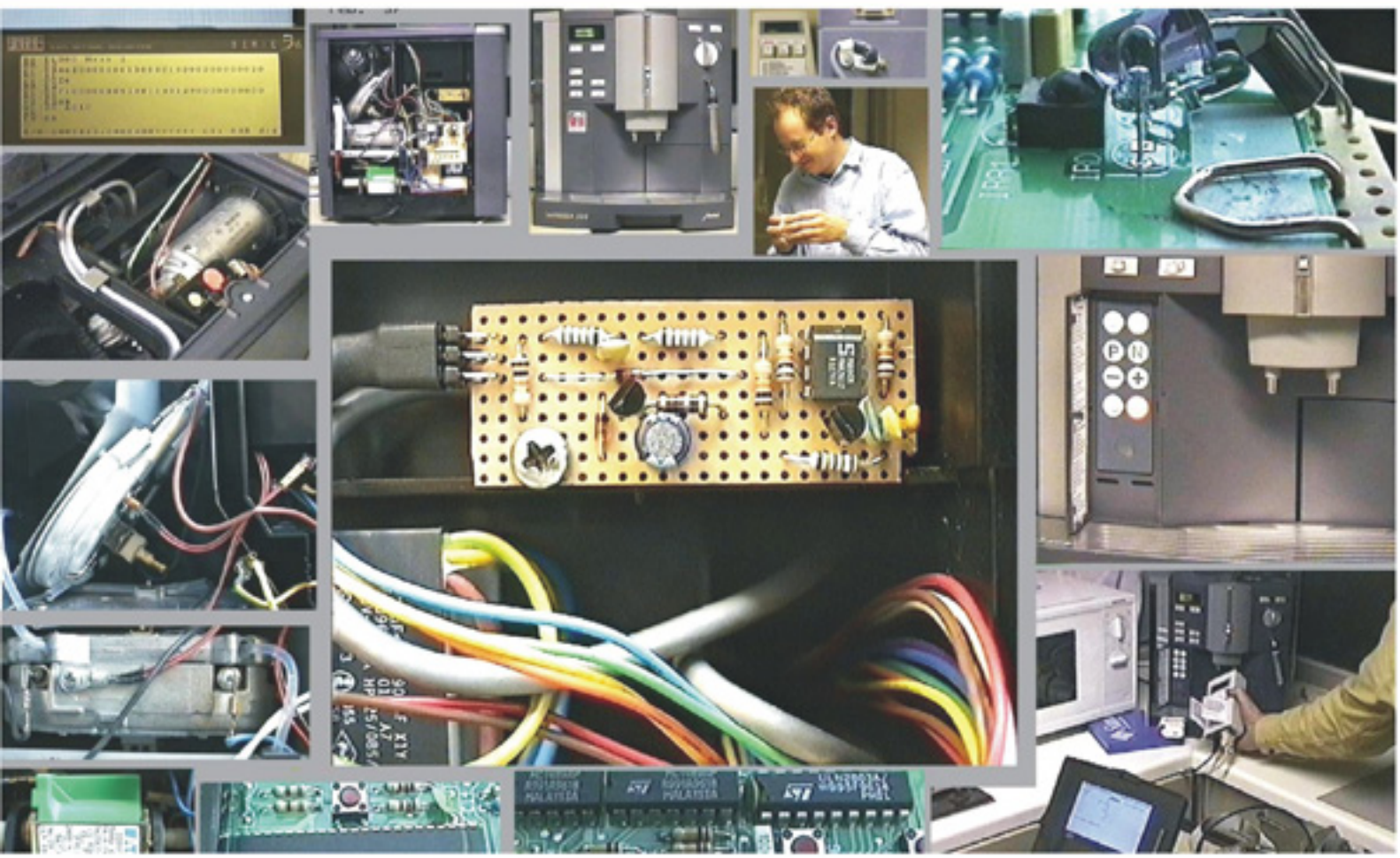
Reverse Engineering

Walter Bell



Training makes a difference...

Global Media
Education For Everyone

First Edition, 2007

ISBN 978 81 89940 16 4

Published by:

Global Media
1819, Bhagirath Palace,
Chandni Chowk, Delhi-110 006
Email: globalmedia@dkpd.com

Table of Contents

Introduction

Reverse Engineering, or “reversing,” is a term that carries various connotations, many of which are negative. But Reverse Engineering does play a vital role in the legitimate process of software development.

Closed-source software ships with two components: the software, and the documentation. Most documentation however, is poor and incomplete. Where then can an end-user turn to get more information? It turns out that all the information you need to use a product exists not in the documentation, but directly in the software itself. All that’s needed are tools and skills to gather the information.

A chain is only as strong as its weakest link. Software developers all depend on at least some pieces of externally prepared software: libraries, compilers, and operating systems. Almost all software is dependent on other software and ensuring the security and reliability of your own software often involves assurance as to the security and reliability of all your software’s dependencies.

Reverse Engineering is the process of examining how software works, and drawing useful conclusions from that data. The “bad guys” can reverse engineer code to find bugs to exploit, so surely the “good guys” can reverse engineer code to find bugs that need fixing. Every day people are reverse engineering software components to gather information that the documentation leaves out. Every day people are deciphering proprietary file formats to maintain compatability and lines of communication. Every day developers are examining their own code, or the code of others, to find and plug holes before they are exploited.

Common uses include:

- recovery of business data from proprietary file formats
- creation of hardware documentation from binary drivers, often for producing Linux drivers from Windows or Macintosh drivers
- enhancing consumer electronics devices
- malware analysis
- malware creation, often involving a search for security holes
- discovery of undocumented APIs that may be useful
- criminal investigation
- copyright and patent litigation
- breaking software copy protection (legally and not), often for games and expensive engineering software

What is Reverse Engineering?

What exactly is reverse engineering? In a general sense, reverse engineering is simply an effort to try and recreate the design of a product by examining the product itself. Reverse engineering is the process of asking “how did they do that?” and then trying to do it yourself. In terms of

software however, reverse engineering involves examining what a piece of software does, and how it does it.

Unfortunately, the title “Disassemblers, Debuggers, System Architectures, File Formats, Compilers and Low-level code Generation” doesn’t roll off the tongue as well as “Reverse Engineering” does.

Ethics

Many people immediately ask themselves “Isn’t reverse engineering something that crackers and criminals do?” and to some degree the answer is yes. Crackers and virus writers all examine existing code to find weaknesses to exploit. However, this is not the only use of reverse engineering tools. For instance, a VCR may be used to illegally duplicate copyrighted movies, but it may also be used to play back precious home videos. Similarly there are two sides to this sword: these tools may certainly be used to hurt, but they can much more often be used to help.

Law

Certain applications of reverse engineering are illegal in certain countries around the world. This book would like to be as informative as possible without landing in legal hot water. As such, examples that may be illegal to test should be avoided.

Computer software is often subjected to patent and copyright considerations. However, certain aspects of an application, such as algorithms, are frequently not covered by copyright. Also, many programs provide inadequate documentation to explain certain features, and reverse engineering is often legal to gain more information in this condition.

For specific issues that may or may not be illegal in your area, consult your lawyer before attempting anything.

Legal Aspects

In the United States, the DMCA (Digital Millenium Copyright Act) is the primary source for information on laws regarding reverse engineering. In other parts of the world, there are other laws that govern Reverse Engineering practices. The reader is advised to seek outside legal counsel before doing anything that might be illegal.

The DMCA Exception

In general, the DMCA bans malicious reverse engineering. However, there is an exeption in the DMCA saying that reverse engeenering can be done under the purposes of inter-operability between software components.

Proprietary Software Data Imprisonment

There is also a law that permits a person to develop a software component that permits them to retrive their data from proprietary software (the linux kernel problem with a proprietary software component they used for the kernel development).

Process

Copyright Work

There are laws about the copyright that someone who reverse-engineers must take care of in open source projects the common approach of this problem is dividing the programmers into 2 groups:

1. The one who disassembles the code of the program/firmware and writes the specifications.
2. The second group that makes a program using these specifications.

Firmware

Normally when you install a program/os you have an end user licence agreement (EULA) that tells you what uses of the software are permited or not, except where such a license is absent.

“Hackers”

Hacking is a term used in popular culture to describe malicious activities of computer users. The movie Hackers was a large influence on bringing the term into common use by romanticising the *Hacker* as an idealistic youth seeking freedom from tyranny.

There are some fantastic books that help to explain what a *real* hacker is like:

- *Hackers*, by Steven Levy
- *The Devouring Fungus*, by Karla Jennings
- *Free as in Freedom*, by Sam Williams
- *Just for Fun*, by Linus Torvalds
- The Cathedral and the Bazaar, by Eric Raymond
- *Code Book*, by Simon Singh
- In The Beginning... Was The Command Line, by Neal Stephenson
- *the cluetrain manifesto*, by Rick Levine, Christopher Locke, Doc Searls, David Weinberger

This hopes to shed some light on what hackers really do, and who they actually are.

Hackers are people who enjoy playing around with computers to make things happen. This often involves circumventing some security aspects of operating systems or applications in order to gain privilaged access.

The first chapter is one of the most important chapters to read. Here the term Hacking is defined, revealing some insight into what hacking really is.

The second covers the history of computing and hackers. This might help correct the false impressions propigated by news media.

The ‘hacking-culture’ follows up next and ‘finally’ the real thing is assesed. (Note: These methods are illegal if used wrongly, yet the method to prevent or ‘cure’ this ‘attack’ is given as well to remain as objective as possible).

I would appreciate it if anyone posts stuff which will help the world to deal with security issues and how to ‘deal’ with hackers (mostly crackers and scriptkiddies).

The Jargon File or New Hackers dictionary defines the term hacker quite nicely. It also does an exceptional job of pointing out that one does not need to be affiliated with computers at all to be considered a hacker.

The excerpt from The Jargon File:

```
hacker: n.
[originally, someone who makes furniture with an axe]
1. A person who enjoys exploring the details of programmable systems
```

```
and how to stretch their capabilities, as opposed to most users, who

prefer to learn only the minimum necessary. RFC1392, the Internet

Users' Glossary, usefully amplifies this as: A person who delights in
having an intimate understanding of the internal workings of a system,
computers and computer networks in particular.
2. One who programs enthusiastically (even obsessively) or who enjoys
programming rather than just theorizing about programming.
3. A person capable of appreciating hack value.
4. A person who is good at programming quickly.
5. An expert at a particular program, or one who frequently does work
using it or on it; as in ?a Unix hacker?. (Definitions 1 through 5 are
correlated, and people who fit them congregate.)
6. An expert or enthusiast of any kind. One might be an astronomy hacker,
for example.
7. One who enjoys the intellectual challenge of creatively overcoming or
circumventing limitations.
8. [deprecated] A malicious meddler who tries to discover sensitive
information by poking around. Hence password hacker, network hacker.
The correct term for this sense is cracker.
```

Maybe it is helpful to note that the people that program the Linux kernel are called “Linux hackers”.

A brief History of Hackers

As long as there have been computers, people were there to ‘hack’ them. But this activity really hit the headlines when the Internet arrived. Yet history teaches us that this event wasn’t an evil thing at all, hackers actually ‘maintain’ the Internet as it should be. It is unimaginable for the computer/Internet society to grow so large without people who were at the cutting edge of technology (hacking internet it’s way up). Just imagine how it would be if there weren’t any hackers... Most technology you use today would cease to exist. Ask yourself: Would your computer be this powerful if it wasn’t put to the edge? Would software be as reliable as they look? Would there be a spiral of new cutting edge innovations?

You can answer all these questions by no. It might seem strange to think positive of hackers, but know that much is done on the edge of society (mostly not in the middle).

The Hacker-Culture

Advanced computer users describe themself as hackers; those who use their skills for malevolent purposes are termed “crackers”. The term crackers is a term that implies breaking things, in the sense of cracking the integrity of a computer system; and they work through cracks in security, like climbing through a crack in a wall; but their main act is breaking into computers for example by figuring out the root password, which is like cracking a safe at a bank. This means that crackers break the law, yet this isn’t enough to get indepth information about the hacker-culture.

In this chapter the main hacker-personalities will be described. In a rather unusual way: the media is used to get to know the real group. This means that you'll be able to understand that some of the people certainly not worth the name hacker...

Terminology

Hack is an onomatopoeic verb describing the noise and actions of chopping at something with a blade (i.e.: *He hacked away at the underbrush with a machete*), or a particularly nasty cough (i.e.: *The chainsmoker hacked up some brown phlegm*), but which also came to describe the act of typing on a typewriter, for the same reason (the annoying, incessant HACK HACK HACK, Ding! *CRASH!* HACK HACK HACK, &c).

From there it became associated not only with the action itself, but also those doing the typing. For example, a "hack"—a bad writer/journalist—would "hack out" a poorly researched or unoriginal story on his typewriter. While the less noisy tapping of computer keyboards began to replace the harsh noise of the typewriter, the old terminology was carried over to the new technology. Thus, the original "hackers," (long before the PC or word processors) were merely called that because they would spend their days "hacking away" at their console keyboards, writing code.

Note: It's worth noting that, when computing was in its infancy and console time at the giant mainframes was scarce, programmers would often hand-write code or type it out on typewriters before they manually plugged it into the machine. Also, the earliest consoles were basically automated typewriters.)

Nowadays, a **hacker** is, within the software development community, any skilled programmer, especially among open-source developers. A **hack** in turn, is a quick-and-dirty patch, fix, or utility which may not be well documented or necessarily reliable, but which gets the job done, whatever that job may be.

Crackers are skilled programmers who exploit the limitations of computer networks, and write up *cracks*—malicious hacks—to automate the dirty work. These hacks/cracks may attempt to break into remote machines e.g. *He hacked into the school's server to increase his phys-ed grade.*, crack passwords (the most useful utility for this is simply called *Crack*), decrypt data, or simply modify proprietary software so the cracker doesn't have to pay for it e.g. *He downloaded a cracked version of Dreamweaver, because he couldn't afford to buy it.*.

Viruses, trojans, and worms are also hacks/cracks of a sort. Crackers are especially fond of *worms*, which spread without user interaction and can be used to create giant, distributed supercomputers that can then be used to attack other computers (the *Code Red* worm used the combined power of infected computers to flood the White House web server, making it inaccessible to regular users for a time).

When a hacker finds a security hole in the software they're using, they hack out some code to patch it. When a cracker finds a hole, they hack out code to exploit it, ideally bringing remote computers under their control.

Script-kiddies use hacks and cracks created by real programmers, but they use their software without really understanding the code that's doing the work. They are generally trying to just impress their friends.

As indicated above, not all hacks are "malware" (malicious code). User-created shell scripts and batch files that automate tasks (like workstation startup, permission settings, data backup, &c.) are hacks too. Hacks are tools. They are hacked out to make someone's life easier, but not necessarily yours.

TV news and, of course, the movie "Hackers" brought the label *hacker* to the attention of the wider public, but failed to acknowledge its broader meaning, instead using it as a buzzword, so as not to confuse the less informed members of their audience (*"Computer programs are written by people? Like books? I thought other computers made them!"*).

Others have latched on to the grit, glamour, and rebellion the buzzword *hacker* invokes; thus, they think of hacking as something of a religion. But, in short, it's nothing more than playing around with computer code. You don't even need to be connected to the internet to do some hacking, just learn a programming language.

Tools

- Reverse Engineering requires a number of specialized tools that are not normally familiar to many computer users (and even many software developers). This section will discuss some of the common tools used by reversers, and will list the tools that are used in examples throughout this .

Notes on the Section

The remainder of the will assume that the reader has a disassembler and a debugger (or alternatively, a debugger that disassembles). The choice of disassembler and debugger are not necessarily important, although the chosen debugger should have the ability to set breakpoints on the execution (most do).

the **dumpbin** utility will be used implicitly to discover details about PE files, including function imports and exports. **readelf** will be used similarly in examples that involve ELF files.

This section will not consider tools such as text editors, code beautifiers, or IDEs, which are useful to reversers, but do not actively help with the reversing process.

Section 1: Background

Disassemblers

This page will discuss disassembler software and techniques. All listed software should include a download site (if it is freeware/Open Source) or a homepage (if it costs money). We aren't going to limit this to only Free tools, although they may be the most popular for starting with Reversing. If the list of Disassemblers becomes too large, the entries may be moved to the appendix.

A further discussion of disassemblers, and the process of disassembly will be covered in later sections.

What is a Disassembler?

In essence, a **disassembler** is the exact opposite of an assembler. Where an Assembler converts code written in an assembly language into binary machine code, a disassembler reverses the process and attempts to recreate the assembly code from the binary machine code.

Since most assembly languages have a one-to-one correspondence with underlying machine instructions, the process of disassembly is relatively straight-forward, and a basic disassembler can often be implemented simply by reading in bytes, and performing a table lookup. Of course, disassembly has its own problems and pitfalls, and they are covered later in this chapter.

Many disassemblers have the option to output assembly language instructions in Intel, AT&T, or (occasionally) HLA syntax

x86 Disassemblers

Windows Disassemblers

For convenience, we will break up Windows-based disassemblers into 2 catagories: **Professional Tools** (which cost money), and **Freeware Tools** which are free.

Professional Tools

IDA Pro

> A professional (read: expensive) disassembler that is extremely powerful, and has a whole slew of features. The downside to IDA Pro is that it costs $439 US for the standard single-user edition. As such, while it is certainly worth the price, this will not consider IDA Pro specifically because the price tag is exclusionary. Two freeware versions do exist;

PE Explorer

is a disassembler that “focuses on ease of use, clarity and navigation.” It isn’t as feature-filled as IDA Pro, but carries a smaller price tag to offset the missing functionality: $130

Freeware Tools

IDA 3.7

This is a DOS GUI tool that behaves very much like IDA Pro, but is considerably more limited. It can disassemble code for the Z80, 6502, Intel 8051, Intel i860, and PDP-11 processors, as well as x86 instructions up to the 486.

http://www.simtel.net/product.php

IDA Pro Freeware 4.1

Behaves almost exactly like IDA Pro, but it only disassembles code for Intel x86 processors, and only runs on Windows. It can disassemble instructions for those processors available as of 2003.

http://www.themel.com/idafree.zip

BORG Disassembler

BORG is an excellent Win32 Disassembler with GUI.

http://www.caesum.com/

HT Editor

An analyzing disassembler for Intel x86 instructions. The latest version runs as a console GUI program on Windows, but there are versions compiled for Linux as well.

http://hte.sourceforge.net/

diStorm64

diStorm is an open source highly optimized stream disassembler library for 80x86 and AMD64.

http://ragestorm.net/distorm/

Linux Disassemblers

Bastard Disassembler

The Bastard disassembler is a powerful, scriptable disassembler for Linux and FreeBSD.

http://bastard.sourceforge.net/

objdump

comes standard, and is typically used for general inspection of binaries. Pay attention to the relocation option and the dynamic symbol table option.

gdb

comes standard, as a debugger, but is very often used for disassembly. If you have loose hex dump data that you wish to disassemble, simply enter it (interactively) over top of something else or compile it into a program as a string like so: char foo[] = {0x90, 0xcd, 0x80, 0x90, 0xcc, 0xf1, 0x90};

lida linux interactive disassembler

an interactive disassembler with some special functions like a crypto analyzer. Displays string data references, does code flow analysis, and does not rely on objdump. Utilizes the Bastard disassembly library for decoding single opcodes.

http://lida.sourceforge.net

ldasm

LDasm (Linux Disassembler) is a Perl/Tk-based GUI for objdump/binutils that tries to imitate the 'look and feel' of W32Dasm. It searches for cross-references (e.g. strings), converts the code from GAS to a MASM-like style, traces programs and much more. Comes along with PTrace, a process-flow-logger.

http://www.feedface.com/projects/ldasm.html

Disassembler Issues

Separating Code from Data

Since data and instructions are all stored in an executable as binary data, the obvious question arises: how can a disassembler tell code from data? Is any given byte a variable, or part of an instruction?

The problem wouldn't be as difficult if data were limited to the .data section of an executable (explained in a later chapter) and if executable code was limited to the .code section of an executable, but this often not the case. Data may be inserted directly into the code section (e.g. jump address tables, constant strings), and executable code may be stored in the data section (although new systems are working to prevent this for security reasons).

Many interactive disassemblers will give the user the option to render segments of code as either code or data, but non-interactive disassemblers will make the separation automatically. Disassemblers often will provide the instruction AND the corresponding hex data on the same line, to reduce the need for decisions to be made about the nature of the code.

The general problem of separating code from data in arbitrary executable programs is equivalent to the halting problem. As a consequence, it is not possible to write a disassembler that will correctly separate code and data for all possible input programs. Reverse engineering is full of such theoretical limitations, although by Rice's theorem all interesting questions about program

properties are undecidable (so compilers and many other tools that deal with programs in any form run into such limits as well).

Lost Information

Much information is lost when a program is compiled. For instance, the names of functions, variables, and labels in the code will all be lost after compilation. C language constructs will often be altered to become more efficient, or easier for the compiler to reproduce in assembly code. Comments in the code all are discarded by the compiler. It will not be possible to determine the difference between code that was written out in-place, code that was written as an inline function, and code that was written up as a C-preprocessor macro. In many cases it will not be possible to determine lexicographical scope of functions or variables. If two files are compiled and linked together, file1.c and file2.c, the delineation between source files will disappear during the linking stage.

Questions

Question 1: Write a simple “Hello world!” (see K&R, chapt 1) program in any compiled language of your choice. Compile this code, and disassemble the resulting executable. Is the resulting assembly code longer or shorter than your original code?

Answers to Questions

Answer 1: The disassembly code will be larger, much larger. My original code—written in C—was about 6 lines long. The resulting disassembly was over 20,000. This phenomenon will be discussed in later chapters about executable file structures.

Decompilers

Akin to Disassembly, **Decompilers** take the process a step further and actually try to reproduce the code in a high level language. Frequently, this high level language is C, because C is simple and primitive enough to facilitate the decompilation process. Decompilation does have its drawbacks, because lots of data and readability constructs are lost during the original compilation process, and they cannot be reproduced. Since the science of decompilation is still young, and results are “good” but not “great”, this page will limit itself to a listing of decompilers, and a general (but brief) discussion of the possibilities of decompilation.

Decompilation: Is It Possible?

In the face of optimizing compilers, it is not uncommon to be asked “Is decompilation even possible?” To some degree, it usually is. Make no mistake, however: there are no perfectly operational decompilers (yet). At most, current Decompilers can be used as simply an aid for the reversing process, with lots of work from the reverser.

Common Decompilers

DCC Decompiler

> Dcc is an excellent theoretical look at decompilation, but currently it only supports small files.
>
> http://www.itee.uq.edu.au/~cristina/dcc.html

Boomerang Decompiler Project

> Boomerang Decompiler is an attempt to make a powerful, retargetable compiler. So far, it only decompiles into C with moderate success.

Reverse Engineering Compiler (REC)

> REC is a powerful “decompiler” that decompiles native assembly code into a *C-like* code representation. The code is half-way between assembly and C, but it is much more readable then the pure assembly is.
>
> http://www.backerstreet.com/rec/rec.htm

ExeToC

> ExeToC decompiler is an interactive decompiler that boasts pretty good results.
>
> http://sourceforge.net/projects/exetoc

Note on the Chapter

Decompilers are considered here as a matter of interest, but a few points need to be considered:

- The field of native code decompilation is in its infancy.
- Current decompilers all require significant input from the user.
- Current decompilers rarely work well with executable files in the size range from “normal” to “large”.
- the steps in dealing with a decompiler are strongly dependant on the particular decompiler being used.

The remainder of this will probably not consider the matter of decompilation any further than this chapter. However, if the reader is interested in pursuing the subject further, they are encouraged to follow one of the links on this page. Links to other decompilers will probably all be added, because there are so few decompilers.

Debuggers

Debuggers are, with the possible exception of a powerful decompiler, a reverser's best friend. A debugger allows the user to step through program execution, and examine various values and actions.

Advanced debuggers often contain at least a rudimentary disassembler, often times hex editing and reassembly features. Debuggers often allow the user to set "breakpoints" on instructions, function calls, and even memory locations.

A **breakpoint** is an instruction to the debugger that allows program execution to be halted when a certain condition is met. for instance, when a program accesses a certain variable, or calls a certain API function, the debugger can pause program execution.

Windows Debuggers

OllyDbg

> OllyDbg is a powerful Windows debugger with a built-in disassembly **and** assembly engine. Has numerous other features including a 0$ price-tag. Very useful for patching, disassembling, and debugging.
>
> http://www.ollydbg.de/

SoftICE

> A *de facto* standard for Windows debugging. SoftICE can be used for *local kernel debugging*, which is a feature that is very rare, and very valuable. SoftICE was taken off the market in April 2006.

WinDBG

> WinDBG is a *free* piece of software from microsoft that can be used for local user-mode debugging, or even *remote* kernel-mode debugging. WinDBG is not the same as the better-known Visual Studio Debugger, but comes with a nifty GUI nonetheless. Comes in 32 and 64 bit versions.
>
> http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx

IDA Pro

> The multi-processor, multi-OS, interactive disassembler, by DataRescue.
>
> http://www.datarescue.com

Linux Debuggers

gdb

the GNU debugger, comes with any normal Linux install. It is quite powerful and even somewhat programmable, though the raw user interface is harsh.

emacs

the GNU editor, can be used as a front-end to gdb. This provides a powerful hex editor and allows full scripting in a LISP-like language.

ddd

the Data Display Debugger. It's another front-end to gdb. This provides graphical representations of data structures. For example, a linked list will look just like a textbook illustration.

strace, ltrace, and xtrace

let you run a program while watching the actions it performs. With strace, you get a log of all the system calls being made. With ltrace, you get a log of all the library calls being made. With xtrace, you get a log of some of the funtion calls being made.

valgrind

executes a program under emulation, performing analysis according to one of the many plug-in modules as desired. You can write your own plug-in module as desired.

NLKD

A kernel debugger.

http://forge.novell.com/modules/xfmod/project/?nlkd

Debuggers for Other Systems

dbx

the standard Unix debugger on systems derived from AT&T Unix. It is often part of an optional development toolkit package which comes at an extra price. It uses an interactive command line interface.

ladebug

an enhanced debugger on Tru64 Unix systems from HP (originally Digital Equipment Corporation) that handles advanced functionality like threads better than dbx.

DTrace

an advanced tool on Solaris that provides functions like profiling and many others on the entire system, including the kernel.

Debugger Techniques

Setting Breakpoints

As previously mentioned in the section on disassemblers, a 6-line C program doing something as simple as outputting “Hello, World!” turns into massive amounts of assembly code. Most people don’t want to sift through the entire mess to find out the information they want. It can even be time consuming just to FIND the information one desires by just looking through. As an alternative, one can choose to set breakpoints to halt the program once it has reached a given point within the program.

For instance, let’s say that in your program, you consistantly experience crashes at one particular section, immediately after closing a message box. You set a breakpoint on all calls to MessageBoxA. You run your program with the breakpoints, and it stops, ready to call MessageBoxA. Stepping line by line through the program and watching the stack, you see that a buffer overflow occurs shortly after.

Hex Editors

Hex editors, while not a very popular tool for reversing, are useful in that they can directly view and edit the binary of a source file. Also, hex editors are very useful when examining the structure of proprietary closed-format data files.

There are many many Hex Editors in existence, so this page will attempt to weed out some of the best, some of the most popular, or some of the most powerful.

Windows Hex Editors

Axe

> suggested by the CVS one-time use camcorder hackers (discussed later).
> http://www.jbrowse.com/products/axe/

HxD (Freeware)

> fast and powerful hex, disk and RAM editor
> http://mh-nexus.de/hxd/

Freeware Hex Editor XVI32

> A freeware hex editor for windows.
> http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm

Catch22 HexEdit

> This is a powerful hex editor with a slew of features. Has an excellent data structure viewer.
>
> http://www.catch22.net/software/hexedit.asp

BreakPoint Hex Workshop

> An excellent and powerful hex-editor, it's usefulness is restricted by the fact that it is not free like some of the other options.
>
> http://www.bpsoft.com/

Tinyhex

> free, does statistics.
> http://www.mirkes.de/en/freeware/tinyhex.php

frhed - free hex editor

> free, open source for Windows.
> http://www.kibria.de/frhed.html

Cygnus Hex Editor FREE EDITION

A very fast and easy-to-use hex editor.
http://www.softcircuits.com/cygnus/fe/

Hexprobe Hex Editor

A professional hex editor designed to include all the power to deal with hex data, particularly helpful in the areas of hex-byte editing, byte-pattern analysis.

http://www.hexprobe.com/hexprobe/index.htm

UltraEdit32

A hex editor/text editor, won “Application of the Year” at 2005 Shareware Industry Awards Conference.

http://www.ultraedit.com/

ICY Hexplorer

A small, lightweight free hex file editor with some nifty features, such as pixel view, structures, and disassembling.

http://www.elektroda.net/download/file1000.html

WinHex

A powerful hex file and disk editor with advanced abilities for computer forensics and data recovery (also used by governments and military)

http://www.x-ways.net/index-m.html

010 Editor

A very powerful and fast hex editor with extensive support for data structures and scripting. Can be used to edit drives and processes.

http://www.sweetscape.com/010editor/

A view of a small binary file in a 1Fh hex editor.

1Fh

A free binary/hex editor which is very fast even while working with large files. It’s the only Windows hex editor that allows you to view files in byte code (all 256-characters).

http://www.4neurons.com/1Fh/

Linux Hex Editors

emacs

along with everything else, emacs obviously includes a hex editor.

xxd and any text editor

produce a hex dump with xxd, freely edit it in your favorite text editor, and then convert it back to a binary file with your changes included

GHex

Hex editor for GNOME.

http://directory.fsf.org/All_Packages_in_Directory/ghex.html

KHexEdit

Hex editor for KDE - a versatile and customizable hex editor. It displays data in hexadecimal, octal, binary and text mode.

BIEW

a viewer of binary files with built-in editor in binary, hexadecimal and disassembler modes. It uses native Intel syntax for disassemble. Highlight AVR/Java/Athlon64/Pentium 4/K7-Athlon disassembler, russian codepages convertor, full preview of formats - MZ, NE, PE, NLM, coff32, elf partial - a.out, LE, LX, PharLap; code navigator and more over.

http://biew.sourceforge.net/en/biew.html

hview

a curses based hex editor designed to work with large (600+MB) files with as quickly, and with little overhead, as possible.

http://tdistortion.esmartdesign.com/Zips/hview.tgz

HT Editor

A file editor/viewer/analyzer for executables. Its goal is to combine the low-level functionality of a debugger and the usability of IDEs.

http://hte.sourceforge.net/

HexCurse

An ncurses-based hex editor written in C that currently supports hex and decimal address output, jumping to specified file locations, searching, ASCII and EBCDIC output, bolded modifications, an undo command, quick keyboard shortcuts etc.

http://www.jewfish.net/description.php?title=HexCurse

hexedit

view and edit files in hexadecimal or in ASCII.
http://www.geocities.com/SiliconValley/Horizon/8726/hexedit.html

Data Workshop

an editor to view and modify binary data; provides different views which can be used to edit, analyze and export the binary data.

http://www.dataworkshop.de/index.html

VCHE

A hex editor which lets you see all 256 characters as found in video ROM, even control and extended ASCII, it uses the /dev/vcsa* devices to do it. It also could edit non-regular files, like hard disks, floppies, CDROMs, ZIPs, RAM, and almost any device. It comes with a ncurses and a raw version for people who work under X or remotely.

http://www.grigna.com/diego/linux/vche/

Notes on the Chapter

It is no exaggeration to say that there are **many** Hex Editors in existence. There is little or no way to recommend one editor over another, and the choice of a hex editor frequently is reduced to a matter of personal preference.

Other Tools

This chapter is going to list some extra tools, that don't fit neatly into the previously mentioned categories. Some of these tools are just as useful as any that we have discussed before.

Other Tools for Windows

Resource Monitors

SysInternals Freeware

> This page has a large number of excellent utilities, many of which are very useful to security experts, network administrators, and (most importantly to us) reversers. Specifically, check out **FileMon, TCPView, RegMon,** and **Process explorer.**
>
> http://www.sysinternals.com

PE File Header dumpers

Dumpbin

> Dumpbin is a program that previously used to be shipped with MS Visual Studio, but recently the functionality of Dumpbin has been incorporated into the Microsoft Linker, link.exe. to access dumpbin, pass /dump as the first parameter to link.exe:

```
link.exe /dump [options]
```

> It is frequently useful to simply create a batch file that handles this conversion:

```
::dumpbin.bat
link.exe /dump %*
```

All examples in this that use dumpbin will call it in this manner.

> Here is a list of usefull features of dumpbin [3]:

```
dumpbin /EXPORTS         displays a list of functions exported from a library
dumpbin /IMPORTS         displays a list of functions imported from other libraries
dumpbin /HEADERS         displays PE header information for the executable
```

> http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_dumpbin_reference.asp

Other Tools for Linux

oprofile
: can be used the find out what functions and data segments are used

subterfugue
: is a tool for playing odd tricks on an executable as it runs. The tool is scriptable in python. The user can write scripts to take action on events that occur, such as changing the arguments to system calls.

  http://subterfugue.org/

lizard
: lets you run a program ***backwards***.
  http://lizard.sourceforge.net/index.html

dprobes
: lets you work with both kernel and user code

biew
: both hex editor and disassembler

ltrace
: shows runtime library call information for dynamically linked executables

Program Creation

The following few chapters deal with “the enemy”, or the entities that create and compile the programs we are trying to analyze. The easiest thing for a reverser to read in the computer world is fully-commented, well-written, high-level source code. Unfortunately, a whole collection of utilities transform that beautiful source into menacing, optimized binary machine code. In essence, a reverser tries to undo what a compiler has done. In order to better understand the job at hand (reversing) it is imperative that the enemy be understood.

This chapter does not mean to be an introduction to programming, simply an overview of some of the most common tools of software production. If the reader knows about compilers and assemblers, this section may safely be skipped.

If you are interested in a language or compiler that hasnt been listed above, it might appear in the following “catch-all” chapter. If you know of a compiler that doesnt fit into one of the above chapters, write about it here as well.

Listing Files

Many compilers contain the ability to output compiled code in *assembly language format*, instead of in machine code. Such an output is frequently referred to as a “listing file”. Listing files are usually very explicit, and they often contain the human-readable names of functions and variables. Listing files also frequently will show where each C code line starts and stops. This is very useful for a beginning reverser, because it is easy to see exactly how different C instructions map to assembly language instructions, and how various C constructs are implemented in

assembly. Assembly listing files will be used in many examples in Section 4, and will appear in other places as well.

Notes on the Text

The rest of the text assumes familiarity with assembly language and high-level concepts. It is not necessary to have an assembler, or a compiler while reading this text, although occasionally they may be considered in questions and examples. Much of the code examples will deal with C or C++, while Section 6 deals exclusively with C# and Java.

Assemblers

Assemblers are significantly simpler than compilers, and are often implemented to simply translate the assembly code to binary machine code via one-to-one correspondence. Assemblers rarely optimize beyond choosing the shortest form of an instruction or filling delay slots.

The following are some basic assembler concepts, followed by a (very incomplete) list of common assemblers:

Assembler Concepts

Assemblers, on a most basic level, translate assembly instructions into machine code bytes with a 1 to 1 correspondance. Assemblers also allow for named variables that get translated into hard-coded memory addresses. Assemblers also translate labels into their relative code addresses.

Assemblers, unlike compilers, do not optimize. The code that comes out of a assembler is equivalent to the assembly instructions that go into the assembler. Some assemblers have high-level capabilities in the form of *Macros.*

(x86) Intel Syntax Assemblers

Because of the pervasiveness of Intel-based I-32 microprocessors in the home PC market, the majority of assembly work done (and the majority of assembly work considered in this) will be x86 based. Many of these assemblers (or new versions of them) can handle I-64 code as well, although this will focus primarily on 32 bit code examples.

MASM

MASM is Microsoft's assembler, an abbreviation for "Macro Assembler." However, many people use it as an acronym for "Microsoft Assembler," and the difference isnt a problem at all. MASM has a powerful macro feature, and is capable of writing very low-level syntax, and pseudo-high-level code with it's macro feature. MASM 6.15 is currently available as a free-download from microsoft, and MASM 7.xx is currently available as part of the microsoft platform DDK.

- MASM writes in Intel Syntax.
- MASM is used by Microsoft to implement some low-level portions of it's Windows Operating systems.
- MASM, contrary to popular belief, has been in constant development since 1980, and is upgraded on a needs-basis.
- MASM has always been made compatable by Microsoft to the current platform, and executable file types.
- MASM currently supports all Intel instruction sets, including SSE2.

Many users love MASM, but many more still dislike the fact that it isnt portable to other systems.

MASM32 Project

The MASM32 project is a collection of software to support development using MASM. The site has an excellent forum with very knowledgable people.

TASM

TASM, Borland's "Turbo Assembler," is a functional assembler from Borland that integrates seamlessly with Borland's other software development tools. Current release version is version 5.0. TASM syntax is very similar to MASM, although it has an "IDEAL" mode that many users prefer. TASM is not free.

NASM

NASM, the "Netwide Assembler," is a portable, retargetable assembler that works on both Windows and Linux. It supports a variety of Windows and Linux executable file formats, and even outputs pure binary. NASM comes with it's own disassembler.

NASM is not as "mature" as either MASM or TASM, but is a) more portable then MASM, b) cheaper then TASM), and c) strives to be very user-friendly.

FASM

FASM, the "Flat Assembler" is an opensource assembler that supports x86, and IA-64 Intel architectures.

(x86) AT&T Syntax Assemblers

AT&T syntax for x86 microprocessor assembly code is not as common as Intel-syntax, but the GNU GAS assembler uses it, and it is the *de facto* assembly standard on linux.

GAS

The GNU Gas Assembler is the default back-end to the GNU GCC compiler suite. As such, GAS is as portable and retargetable as GCC is. However, GAS uses the AT&T syntax for it's instructions, which some users find to be less readable then Intel syntax. As a result, assembly code written inline in a C code file for GCC must also be written in GAS syntax.

GAS is developed specifically to be used as the GCC backend. GCC always feeds GAS syntactically-correct code, so GAS often has minimal error checking.

GAS is available either a) in the GCC package, or b) in the GNU BinUtils package. [4]

Other Assemblers

HLA

HLA, short for “High Level Assembler” is a project spearheaded by Randall Hyde to create an assembler with high-level syntax. HLA works as a front-end to other compilers such as MASM, NASM, and GAS. HLA supports “common” assembly language instructions, but also implements a series of higher-level constructs such as loops, if-then-else branching, and functions. HLA comes complete with a comprehensive standard library.

Since HLA works as a front-end to another assembler, the programmer must have another assembler installed to assemble programs with HLA. HLA code output therefore, is as good as the underlying assembler, but the code is much easier to write for the developer. The high-level components of HLA may make programs less efficient, but that cost is often far out weighed by the ease of writing the code. HLA high-level syntax is very similar in many respects to Pascal, (which in turn is itself similar in many respects to C), so many high-level programmers will immediately pick up many of the aspects of HLA.

Here is an example of some HLA code:

```
mov(src, dest);  //C++  style comments
pop(eax);
push(ebp);
for(mov(0, ecx); ecx < 10; inc(ecx)) do
mul(ecx);
endfor;
```

Some disassemblers and debuggers can disassemble binary code into HLA-format, although none can faithfully recreate the HLA macros.

Compilers

This chapter is designed to teach people the basics of what a **compiler** does (how a compiler does what it does is a very advanced topic that will need to be the subject of another). We will start by explaining the basic vocabulary of compilers, followed by an overview of the basic structure of a compiler.

Key Words

Compiler

> A compiler is a program that converts instructions from one language into equivalent instructions in another language. There is a common misconception that a compiler always directly converts a high level language into machine language, but this isnt always the case. Many compilers convert code into assembly language, and a few even convert code from one high level language into another. Common examples of compiled languages are: C/C++, Fortran, Ada, and Visual Basic.

Interpreter

> An interpreter is a program that executes a file of instructions in *human readable form.* such programs or "scripts" are not compiled, but are instead interpreted at runtime. The process of interpreting a script every time it is executed takes more time then running a compiled script, but the trade-off is ease of use. Common examples of interpreted languages are: Perl, Python, Lisp, and PHP.

Virtual Machine

> a virtual machine is a program that executes bytecode on a local machine. Since bytecode is not machine-dependant, only the virtual machine needs to be adapted to a target machine for the bytecode to run.

Source Language

> The source language is what the compiler "compiles." for instance, a C compiler compiles the C language.

Intermediate Representation

> When a compiler receives an input code file in the source language, it performs several steps. First, the file is read in and *tokenized:* parts of the code are changed to tokens, and those tokens are arranged internally in such a fashion as to help the compiler with it's other tasks.

Target Language

The target language is what the compiler is supposed to produce. A C compiler frequently sets it's target language to be either assembly language, or native machine code, for instance.

Target Platform

a subsection of compilers, called a "cross-compiler" is a program that takes high-level code input, and outputs instructions (usually in assembly or machine code) for a machine that is different from the machine the compiler runs on. For instance, a developer on an Intel machine may write code to be used on a Sparc target platform.

Front end: Source to Intermediate Representation

The front end of a compiler is module that reads in the source code data, tokenizes it, and converts the code into an intermediate representation. In a standard layered approach to compiler design, the front end encompasses the "Lexical Analyzer" and the "Parser" modules of the compiler.

Some common Front ends are produced by Lex and Yacc (and variants).

Intermediate representations vary for each compiler, but frequently take the shape of either a tree or an instruction stack.

Back end: Target Code Generation

Once the input file has been scanned and parsed into the intermediate representation, the code generator begins it's job of outputting to the target code. Code generators may be either a passive translater, or may be an active optimizing generator.

For a discussion of optimizations that occur during Code Generation, see Reverse Engineering/Code Optimization, and the chapters on Interleaving and Unintuitive Instructions.

- Aho, Alfred V. et al. "Compilers: Principles, Techniques and Tools," Addison Wesley, 1986. ISBN: 0321428900

C Compilers

The purpose of this chapter is to list some of the most common C and C++ Compilers in use for developing *production-level* software. There are many many C compilers in the world, but the reverser doesnt need to consider all cases, especially when looking at professional software. This page will discuss each compiler's strengths and weaknesses, it's availablity (download sites or cost information), and it will also discuss how to generate an assembly listing file from each compiler.

Microsoft C Compiler

The Microsoft C compiler is available from Microsoft for free as part of the Windows Server 2003 SDK. It is the same compiler and library as is used in MS Visual Studio, but doesnt come with the fancy IDE. The MS C Compiler has a very good optimizing engine. It Compiles C and C++, and has the option to Compile C++ code into MSIL (See Section 6 for more on MSIL) instead of native bytecode.

Microsoft's compiler only supports Windows systems, and Intel-compatible 16/32/64 bit architectures.

The MS C compiler is **cl.exe** and the linker is **link.exe**

Listing Files

In this , cl.exe is frequently used to produce assembly listing files of C source code. To produce an assembly listing file yourself, use the syntax:

```
cl.exe /Fa<assembly file name> <C source file>
```

The "/Fa" switch is the command-line option that tells the compiler to produce an assembly listing file.

For example, the following command line:

```
cl.exe /FaTest.asm Test.c
```

Would produce an assembly listing file named "Test.asm" from the C source file "Test.c". Notice that there is no space between the /Fa switch and the name of the output file.

FSF GCC Compiler

This compiler is available for most systems and it is free. Many people use it exclusively so that they can support many platforms with just one compiler to deal with. The GNU GCC Compiler is the *de facto* standard compiler for Linux and Unix systems. It is retargetable, allowing for many input languages (C, C++, Obj-C, Ada, Fortran, etc...), and supporting multiple target OS's and architectures. It optimizes well, but has a non-agressive I-32 code generation engine.

The GCC frontend program is "gcc" ("gcc.exe" on windows) and the associated linker is "ld" ("ld.exe" on windows).

Listing Files

To produce an assembly listing file in GCC, use the following commandline syntax:

```
gcc.exe -S <C sourcefile>.c
```

For example, the following commandline:

```
gcc.exe -S test.c
```

Will produce an assembly listing file named “test.s”. Assembly listing files generated by GCC will be in GAS format. GCC listing files are frequently not as well commented and laid-out as are the listing files for cl.exe.

Intel C Compiler

This compiler is used only for x86, x86-64, and IA-64 code. It is available for both Windows and Linux. The Intel C compiler was written by the people who invented the original x86 architecture: Intel. Intel’s development tools generate code that is tuned to run on Intel microprocessors, and is intended to squeeze every last ounce of speed from an application. AMD I-32 compatable processors are not guaranteed to get the same speed boosts because they have different internal architectures.

Metrowerks CodeWarrior

This compiler is commonly used for classic MacOS and for embedded systems. If you try to reverse-engineer a piece of consumer electronics, you may encounter code generated by Metrowerks CodeWarrior.

Green Hills Software Compiler

This compiler is commonly used for embedded systems. If you try to reverse-engineer a piece of consumer electronics, you may encounter code generated by Green Hills C/C++.

The Java Compiler

The **Java** language, created by Sun Microsystems, is compiled into an intermediate representation called a “bytecode,” and the bytecode is then executed on a virtual machine. While Java was created by Sun, and isnt currently ISO/ANSI standardized, there are some other Java compilers and Virtual Machines worth considering. The Java Virtual Machine (JVM) and the Java SDK are available from Sun as a free download.

Note: Reverse Engineering of java bytecode files will be discussed in later chapters.

Obtaining Java

The Java Software Development Kit (SDK) is available as a free download from Sun. The Development kit is a separate download from the Java VM. http://java.sun.com/

javac: The Java Compiler

The java source code compiler, **javac.exe** (or just “javac” on linux) compiles .java files into the bytecode .class files. The structure of .class files will be discussed in later chapters.

java: The Java Virtual Machine

The java virtual machine, **java.exe** (or just “java” on linux) executes .class files. .Class files cannot be executed without a java virtual machine available, but Sun’s JVM is not the only option.

msjvm: The Microsoft Java Virtual Machine

The microsoft Java Virtual machine was an attempt by microsoft to enable first-party support for java programs in Internet Explorer. Due to security and legal problems, microsoft has abandoned the MSJVM, and offers a utility to uninstall the software.

Other Java Implementations

GJC

Japhar
: An opensource Java Virtual Machine.
http://www.hungry.com/old-hungry/products/japhar/

Microsoft Java SDK

The Microsoft Java development environment. Offered as a free download.

http://msdn.microsoft.com/visualj/prodinfo/new.asp

Java Bytecode Compilers for Other Languages

JPython

Compiles Python to Java-style bytecode.
http://www.jpython.org/

Bytecode to Native Code Compilers

The C# Compiler

The .NET framework is Microsoft's answer to Sun's Java programming language. In many respects, .NET is more secure and more advanced then Java, although the former is much less wide-spread then the latter (at the time of this writing). Microsoft does provide, free of charge, the de facto .NET implementation on Windows systems, although Open source implementations have been cropping up for Linux as well.

Note: Reverse Engineering of .NET bytecodes will be discussed in detail in Reverse Engineering/Reversing Bytecode

How to Obtain .NET

The .NET framework is available as a free-download from Microsoft for Windows users. Users of Unix/Linux/Mac can often find a good alternative Open-Source implementation.

C# compiler

The .NET framework comes with the Microsoft C# command-line compiler installed. The compiler is labled **csc.exe** and resides in a folder with lots of other useful tools for .NET software development. Some of these tools will be considered further in Reverse Engineering/Reversing Bytecode

.NET Framework

Here is a list of terms that the reader should become familiar with if they are going to be doing any work with .NET:

.NET framework

The .NET Framework is the set of tools (compilers, and otherwise), standard libraries, and file associations that enable programs written in MSIL to run.

MSIL

The “Microsoft Intermediate Language” is the name that Microsoft uses for the .NET bytecode.

CLI

The “Common Language Runtime” is the official runtime library and environment of the .NET framework. Ideally, the framework on every platform should conform to the standard. The “Common Language” aspect of the name stems from the fact that *any* .NET language (Visual Basic.NET, C#, ASP.NET, J#.NET, etc...) all compile to CLI bytecode, and can easily inter-operate.

For a further discussion of .NET and CLI, see the chapter on .NET reversing in Reverse Engineering/Reversing Bytecode

Managed C++

In addition to compiling C# code into the CLI, the Microsoft C# Compiler can also compile C++ code into the CLI as well. Microsoft has provided a number of extensions to C++ that can only be compiled into CLI, and are not part of the official C++ standard. These extensions are collectively known as **Managed C++**. Because both C# and Managed C++ are both compiled into the same CLI representation, and the same bytecode, we don’t need to discuss the distinction any further.

Other Implementations

DotGNU

An opensource .NET implementation.
http://www.gnu.org/projects/dotgnu/

DotGNU Portable.NET

An offshot of the DotGNU Project, aims to build a set of tools (compiler, disassembler, etc.) for DotGNU.

http://www.southern-storm.com.au/portable_net.html

Mono

An opensource .NET implementation for Linux, Solaris, Mac OS X, Windows, and Unix systems.

http://www.go-mono.com/

Microsoft Shared-Source CLI

> A “shared source” implementation of a working C# compiler and .NET runtime that compiles on FreeBSD and MacOSX (and Windows, of course).
>
> http://www.microsoft.com/downloads/details.aspx?familyid=3a1c93fa-7462-47d0-8e56-8dd34c6292f0&displaylang=en

Other Compilers

This chapter will contain a listing of compilers for languages that do not warrant their own chapters. Some language implementations are significantly unique in certain aspects that they do deserve their own discussion. In the event that a section on any given language becomes large enough, it should probably be separated out into it’s own chapter. If no languages get listed here, perhaps we will just delete the entire chapter.

Perl

Using the utility “perlcc” program (supplied with the perl interpreter), a user can optionally attempt to compile the perl code into any number of other forms. These other forms include C source code (although the C code is very hard to follow), Native code, and Perl Bytecode.

The perlcc C code output often consists of a series of symbol tables, and calls to internal perl functions, so the reverser would simply not find these to be of much use. The Perl Bytecode however is a little more interesting, but still not easy to read by any stretch of the imagination.

The entire Perl cross-compiler suite is listed as being “very experimental”, so only advanced users should put any stock in the process.

To compile a perl script into bytecode:

```
perlcc -B <filename>
```

The bytecode *should* be self-executing, or it *should* be able to run by passing it as an argument to perl.

To compile a perl script into C:

```
perlcc -O <filename>
```

OR:

```
perlcc -c <output file> <input file>
```

The -O option uses the “optimized C backend” which is considered to be the most experimental component of the entire suite.

It is unlikely that too much Perl will be considered throughout the rest of this book.

Operating Systems and Files

Operating Systems

Most software runs on top of an Operating System. Be it Windows or Linux, Unix or Solaris, just about every computer has an OS that needs to be interfaced with. It is important to note, that software needs to interact with the underlying operating system to do many tasks, including file system I/O, user interfacing, multi-tasking, etc... It then becomes a great asset to the reverser to know where and how a program is interfacing with the underlying OS so that certain actions can be tracked. Here are some of the operating systems that will be considered in this :

- Windows
- Linux Mac OS X
- ThreadX
- Operating systems that may eventually be considered in this , should the need (or the relevant information) arise:
- other UNIX and UNIX-like systems
- VxWorks
- TRON
- Symbian OS
- QNX

Fundamentals: the divide between application and OS

It’s important to understand at this point that all modern operating systems have a strict dividing line between user applications and the operating system itself: you may have heard the term “kernel” already. In short, the “kernel” of an operating system is ‘protected’ from applications by functionality implemented on the processor. User applications are not permitted to edit operating system memory or have access to hardware, without passing through a strictly controled set of channels and API calls. Conversely, anything operating in the kernel has complete control over the local machine.

This model of operation has allowed computers to make huge leaps in functionality, since the days when user applications had as much control as operating system. For example, it is now possible for a multi-user environment on computers, each separated from each other. Memory and processor resources can also be fairly allocated by the system as needed.

Executable File Formats

Compilers turn source code into binary modules. Linkers combine various modules to form OS-dependent executable files and libraries. Loaders move those executables into memory, and set up an execution environment. By understanding how different executable file formats are set up, and

how each OS loads them into memory, reversers gain valuable insight into the target program. Here are some topics that we will cover in this section:

- PE files
- ELF files
- Other Executable File Formats

This chapter will be a discussion of some general concepts associated with dynamic linking:

- Dynamic Libraries

Note on the Section

Many executable file formats are significantly outdated that they don't warrant their own chapters. This book will primarily consider Windows NT 5.0 or greater, and Linux Kernel series 2.4 or greater. File formats that are not used on these platforms can be included in the "other" section.

Note on the Text

The remainder of this book generally assumes a familiarity with at least one executable file format, and the ability to list function imports/exports, and identify the address of these functions in the file. Also, the reader should be able to recognize standard module entry sequences, and the origin of "user code."

For systems that allow dynamic linking, the reader should have a vague familiarity with the linking mechanism.

Microsoft Windows

The **Windows operating system** is a popular target for reversers for one simple reason: the OS itself, and most applications for it are not Open Source. Most software on a Windows machine does not have source code, and most pieces have inadequate, or non-existent documentation. Occasionally, the only way to know precisely what a piece of software does (or for that matter, to determine whether a given piece of software is malicious or legitimate) is to reverse it, and examine the results.

Windows Versions

Windows operating systems can be easily divided into 2 categories: Win9x, and WinNT.

Windows 9x

The Win9x kernel was originally written to span the 16bit - 32bit divide. Operating Systems based on the 9x kernel are: Windows 95, Windows 98, and Windows ME. Win9x Series operating systems are known to be prone to bugs and system instability. The actual OS itself was a 32 bit extension of MS-DOS, it's predecessor. An important issue with the 9x line is that they were all based around using the ASCII format for storing strings, rather than unicode.

Development on the Win9x kernel ended with the release of Windows XP.

Windows NT

The WinNT Kernel series was originally written as enterprise-level server and network software. WinNT stresses stability and security far more than Win9x Kernels did (although it can be debated whether that stress was good enough). It also handles all string operations internally in unicode, giving more flexibility when using different languages. Operating Systems based on the WinNT kernel are: Windows NT 4, Windows NT 5(Windows 2000), Windows XP, and the Windows Server 2k Series. Most future Microsoft operating system products are based on NT in some shape or form.

Virtual Memory

32 bit WinNT allows for a maximum of 4Gb of virtual memory space, divided into "pages" that are 4096 bytes by default. Pages not in current use by the system or any of the applications may be written to a special section on the harddisk known as the "paging file." Use of the paging file may increase performance on some systems, although high latency for I/O to the HDD can actually reduce performance in some instances.

System Architecture

The Windows architecture is heavily layered. Function calls that a programmer makes may be redirected 3 times or more before any actions are actually performed. There is an unignorable penalty for calling Win32 functions from a user-mode application. However, the upside is equally unignorable: code written in higher levels of the windows system are much easier to write. Complex operations that involve initializing multiple data structures and calling multiple subfunctions can be performed by calling only a single higher-level function.

The Win32 API comprises 3 modules: KERNEL, USER, and GDI. KERNEL is layered on top of NTDLL, and most calls to KERNEL functions are simply redirected into NTDLL function calls. USER and GDI are both based on WIN32K (a kernel mode module, responsible for the windows "look and feel"), although USER also makes many calls to the more-primitive functions in GDI. This and NTDLL both provide an interface to the Windows NT kernel, NTOSKRNL (see further below).

NTOSKRNL is also partially layered on HAL (Hardware Abstraction Layer), but this interaction will not be considered much in this book. The purpose of this layering is to allow processor variant issues (Such as location of resources) to be made separate from the actual kernel itself. A slightly different system configuration thus requires just a different HAL module, rather than a completely different kernel module.

System calls and interrupts

After filtering through different layers of subroutines, most API calls require interaction with part of the operating system. Services are provided via 'software interrupts', traditionally through the "int 0x2e" instruction. This switches control of execution to the NT executive / kernel, where the request is handled. It should be pointed out here, that the stack used in kernel operation is different to the user mode stack. This provides an added layer of protection between kernel and user. Once the function completes, control is returned back to the user application.

Both Intel and AMD provide an extra set of instructions to allow faster system calls, the "SYSENTER" instruction from Intel and the SYSCALL instruction from AMD.

Win32 API

Both WinNT and Win9x Systems utilize the Win32 API. however, the WinNT version of the API has more functionality, and security constructs, as well as unicode support. Most of the Win32 API can be broken down into 3 separate components, that each perform a separate task.

kernel32.dll

Kernel32.dll, home of the KERNEL subsystem, is where non-graphical functions are implemented. Some of the APIs located in KERNEL are: The Heap API, the Virtual Memory API, File I/O API, the Thread API, the System Object Manager, and other similar system services. Most of the functionality of kernel32.dll is implemented in ntdll.dll, but in

undocumented functions. Microsoft prefers to publish documentation for kernel32 and guarantee that these APIs will remain unchanged, and then put most of the work in other libraries, which are then not documented.

gdi32.dll

gdi32.dll is the library that implements the GDI subsystem, where primitive graphical operations are performed. GDI diverts most of its calls into WIN32K, but it does contain a manager for GDI objects, such as pens, brushes and device contexts. The GDI object manager and the KERNEL object manager are completely separate.

user32.dll

the USER subsystem is located in the user32.dll library file. This subsystem controls the creation and manipulation of USER objects, which are common screen items such as windows, menus, cursors, etc... USER will set up the objects to be drawn, but will perform the actual drawing by calling on GDI (which in turn will make many calls to WIN32K) or sometimes even calling WIN32K directly. USER utilizes the GDI Object Manager.

Native API

The native API, hereby referred to as the NTDLL subsystem is a series of undocumented API function calls that handle most of the work performed by KERNEL. It has been speculated by many that by tapping into NTDLL directly, programs could be spared a certain amount of redirection, and have a performance increase. However, the NTDLL function calls and data structures are usually more complicated then the corresponding KERNEL functions and data structures, so gains are hard to measure. Also, without the added error checking, and the proper calls into kernel mode, the application risks producing errors that are crippling to the system. Microsoft also does not guarantee that the native API will remain the same between different versions, as windows developers modify the software. This gives the risk of native API calls being removed or changed without warning, breaking software that utilizes it.

ntdll.dll

The NTDLL subsystem is located in ntdll.dll. this egnimatic library contains many API function calls, that all follow a particular naming scheme. Each function has a prefix: Ldr, Ki, Nt, Zw, Csr, dbg, etc... and all the functions that have a particular prefix all follow particular rules.

The “official” native API is usually limited only to functions whose prefix is Nt or Zw. These calls are in fact the same: the relevant Export entries map to the same address in memory. Thus there is not read difference, although the reason for the double-mapping results from ntdll’s dual purpose: it is used to provide function calls in both kernel and user space. User applications are encouraged to use the Nt* calls, while kernel callers are supposed to use the Zw* calls. The origin of the prefix “Zw” is unknown.

In actual implementation, the Nt / Zw calls merely load two registers with values required to describe a native api call, and then execute a software interrupt.

Most of the other prefixes are obscure, but the known ones are:

- RTL stands for “Run Time Library”, calls which help functionality at runtime (such as RtlAllocateHeap)
- CSR is for “Client Server Runtime”, which represents the interface to the win32 subsystem located in csrss.exe
- DBG functions are present to enable debugging routines and operations
- LDR provides the ability to manipulate and retrieve data from shared libraries and other module resources

User Mode Versus Kernel Mode

Many of the other functions in NTDLL are usable, but not to application writers. Developers working on writing device drivers for window are frequently **only** allowed to use the Kernel-mode functions in NTDLL because device drivers operate at Kernel-level. As such, Microsoft provides documentation on many of the APIs with prefixes other then Nt and Zw with the Microsoft Server 2003 Platform DDK. The DDK (Driver Development Kit) is available as a free download.

ntoskrnl.exe

This module is the Windows NT “’Executive’”, providing all the functionality required by the native API, as well as the kernel itself, which is responsible for maintaining the machine state. By default, all interrupts and kernel calls are channeled through ntoskrnl in some way, making it the single most important program in windows itself. Many of its functions are exported (all of which with various prefixes, a la NTDLL) for use by device drivers. It’s not advised to try to call these routines from user mode, and the IMAGE_FILE_SYSTEM flag is set in the file’s PE Header, preventing applications from trying this. Some functions from NTOSKRNL may be considered in later examples.

Win32K.sys

This module is the “Win32 Kernel” that sits on top of the lower-level, more primitive NTOSKRNL. WIN32K is responsible for the “look and feel” of windows, and many portions of this code have remained largely unchanged since the Win9x versions. This module provides many of the specific instructions that cause USER and GDI to act the way they do. It’s responsible for translating the API calls from the USER and GDI libraries into the pictures you see on the monitor.

With the coming release of windows “Vista”, it is rumoured that the functionality of Win32K.sys will be taken out of kernel space and placed back into usermode, where it is safer and more isolated.

Win64 API

With the advent of 64-bit processors, 64-bit software is a necessity. As a result, the Win64 API was created to utilize the new hardware. It is important to note that the format of many of the function calls are identical in Win32 and Win64, except for the size of pointers, and other data types that are specific to 64-bit address space.

Differences

Windows Vista

Microsoft has announced a new version of it's Windows operation system, named "Windows Vista." Windows Vista may be better known by it's development code-name "Longhorn." Microsoft claims that Vista has been written largely from the ground up, and therefore it can be assumed that there are fundamental differences between the Vista API and system architecture, and the APIs and architectures of previous Windows versions.

Windows CE, and other versions

Windows CE is the Microsoft offering on small devices. it largely uses the same Win32 API as the desktop systems, although it has a slightly different architecture. Some examples in this book may consider WinCE.

"NonExecutable Memory"

Recent windows service packs have attempted to implement a system known as "Non-executable memory" where certain pages can be marked as being "non-executable". The purpose of this system is to prevent some of the most common security holes by not allowing control to pass to code inserted into a memory buffer by an attacker. For instance, a shellcode loaded into an overflowed text buffer cannot be executed, stopping the attack in its tracks. The effectiveness of this mechanism is yet to be seen, however.

COM and Related Technologies

COM, and a whole slew of technologies that are either related to COM or are actually COM with a fancy name, is another factor to consider when reversing Windows binaries. COM, DCOM, COM+, ActiveX, OLE, MTS, and Windows DNA are all names for the same subject, or subjects, so similar that they may all be considered under the same heading. In short, COM is a method to export Object-Oriented Classes in a uniform, *cross-platform* and *cross-language* manner. In essence, COM is .NET, version 0 beta. Using COM, components written in many languages can export, import, instantiate, modify, and destroy objects defined in another file, most often a DLL.

This book will attempt to show some examples of COM files, and the reversing challenges associated with them, although the subject is very broad, and may elude the scope of this book (or at least the early sections of it). The discussion may be part of an “Advanced Topic” found in the later sections of this book.

Due to the way that COM works, a lot of the methods and data structures exported by a COM component are difficult to perceive by simply inspecting the executable file. Matters are made worse if the creating programmer has used a library such as ATL to simplify their programming experience. Unfortunately for a reverse engineer, this reduces the contents of an executable into a “Sea of bits”, with pointers and data structures everywhere.

Remote Procedure Calls (RPC)

Linux

The **Linux operating system** is open source, but at the same time there is so much that constitutes “Linux” that it can be difficult to stay on top of all aspects of the system. Here we will attempt to boil down some of the most important concepts of the Linux Operating System, especially from a reverser’s standpoint

System Architecture

The concept of “Linux” is mostly a collection of a large number of software components that are based off the GNU tools and the Linux kernel. Linux is itself broken into a number of variants called “distros” which share some similarities, but may also have distinct peculiarities. In a general sense, all Linux distros are based on a variant of the Linux kernel. However, since each user may edit and recompile their own kernel at will, and since some Distros may make certain edits to their kernels, it is hard to proclaim any one version of any one kernel as “the standard”. Linux kernels are generally based off the philosophy that system configuration details should be stored in aptly-named, human-readable (and therefore human-editable) configuration files.

The linux Kernel implements much of the core API, but certainly not all of it. Much API code is stored in external modules (although users have the option of compiling all these modules together into a “Monolithic Kernel”).

On top of the Kernel generally runs one or more **shells**. Bash is one of the more popular shells, but many users prefer other shells, especially for different tasks.

Beyond the shell, Linux distros frequently offer a GUI (although many distros do not have a GUI at all, usually for performance reasons).

Since each GUI often supplies it’s own underlying framework and API, certain graphical applications may run on only one GUI. Some applications may need to be recompiled (and a few completely rewritten) to run on another GUI.

Configuration Files

Shells

Here are some popular shells:

Bash

> An acronym for “Bourne Again SHell.”

Bourne

A precursor to Bash.

Csh

C Shell

Ksh

Korn Shell

TCsh

A Terminal oriented Csh.

Zsh

Z Shell

GUIs

Some of the more-popular GUIs:

KDE

K Desktop Environment

GNOME

GNU Network Object Modeling Environment

Debuggers

gdb

The GNU Debugger. It comes pre-installed on most linux distributions and is primarily used to debug ELF executables. manpage

winedbg

A debugger for Wine, used to debug Win32 executables under linux. manpage

File Analyzers

strings

Finds printable strings in a file. When, for example, a password is stored in the binary itself (defined statically in the source), the string can then be extracted from the binary without ever needing to execute it. manpage

file

Determines a file type, useful for determining whether an executable has been stripped and whether it’s been dynamically (or statically) linked. manpage

Mac OS X

Apple Computer's **Mac OS X** is the standard Operating System used on Apple Macintosh computers. Other operating systems, primarily Linux, have been ported onto Mac Hardware, and there has been some effort to illegally port OS X onto non-Mac Intel-based hardware, but neither of these efforts has attained the kind of popularity that the "standard bundle" has attained.

Mac OS X has been critically acclaimed by many people in the computer world as being both beautiful and easy to use. OS X is built on a BSD and Mach core but has a certain amount of software that is Mac-specific.

Many Mac users assume that Macs are invulnerable to security threats, whereas some experts in the field seem to think that Macs are as vulnerable as any other system. This chapter will talk about OS X system architecture.

PowerPC

Historically, Macs have run on Motorola 68K-series microprocessors, then IBM PowerPC microprocessors, so many old and current versions of Mac software will only be examinable with 68K and PowerPC disassemblers and debuggers. Recently, however, Apple has switched almost all of its lineup (minus the Xserve, which will ship this October) to the Intel archtecture. Mac software, therefore, is now being released in a Universal Binary format with both PowerPC and Intel code. Such software should be explorable with Intel disassemblers and debuggers.

Architecture

All builds of Mac OS X (OS X) are built on top of a BSD base (using FreeBSD, NetBSD, and OpenBSD), the Mach microkernel, and core system services. Apple has instituted a layered approach to the software design, so that differing kernel versions will not affect the "look and feel" of the Mac.

Older Versions

ThreadX

ThreadX is an embedded OS commonly found in consumer gadgets. For example, you may encounter it while reverse engineering a digital camera or wireless access point.

An example of an ongoing reverse engineering project that you can participate in involves the CVS® One-time-use digital video camera made by Puredigital.

PE Files

PE files are the standard executable file format on Windows NT. PE files are broken down into various sections that can be examined. The purpose of a It is assumed that the reader has read the section on **Virtual Memory** at the Windows page.

Relative Virtual Addressing (RVA)

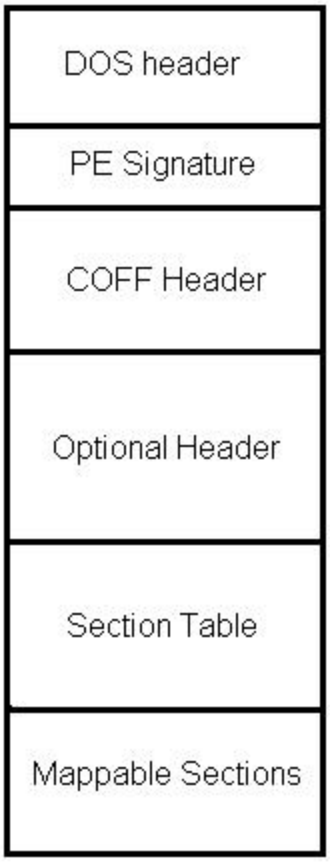
In a Windows environment, executable modules can be loaded at any point in memory, and are expected to run without problem. To allow multiple programs to be loaded at seemingly random locations in memory, PE files have adopted a tool called RVA: Relative Virtual Addresses. RVA's assume that the "Base address" of where a module is loaded into memory is not known at compile time. So, PE files describe the location of data in memory as an *offset from the base address*, wherever that may be in memory.

Some processor instructions require the code itself to directly identify where in memory some data is. This is not possible when the location of the module in memory is not known at compile time. The solution to this problem is described in the section on "Relocations".

It is important to remember that the addresses obtained from a disassembly of a module will not always match up to the addresses seen in a debugger as the program is running.

File Format

The PE portable executable file format includes a number of informational headers, and is arranged in the following format:



The basic format of a Microsoft PE file

MS-DOS header

Open any Win32 binary executable in a hex editor, and note what you see: The first 2 letters are **always** the letters “MZ”. To some people, the first few bytes in a file that determine the type of file are called the “magic number,” although this book will not use that term, because there is no rule that states that the “magic number” needs to be a single number. Instead, we will use the term File ID Tag, or simply, File ID. Sometimes this is also known as File Signature.

After the File ID, the hex editor will show several bytes of either random-looking symbols, or whitespace, before the human-readable string “This program cannot be run in DOS mode”.

What is this?

```
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..............
00000010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00000020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000030  00 00 00 00 00 00 00 00 00 00 00 00 D8 00 00 00  ................
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ........!..L.!Th
00000050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00000060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00000070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00000080  26 C3 8E 85 62 A2 E0 D6 62 A2 E0 D6 62 A2 E0 D6  &...b...b...b...
00000090  E1 BE EE D6 6F A2 E0 D6 62 A2 E0 D6 6D A2 E0 D6      o   b   m
```

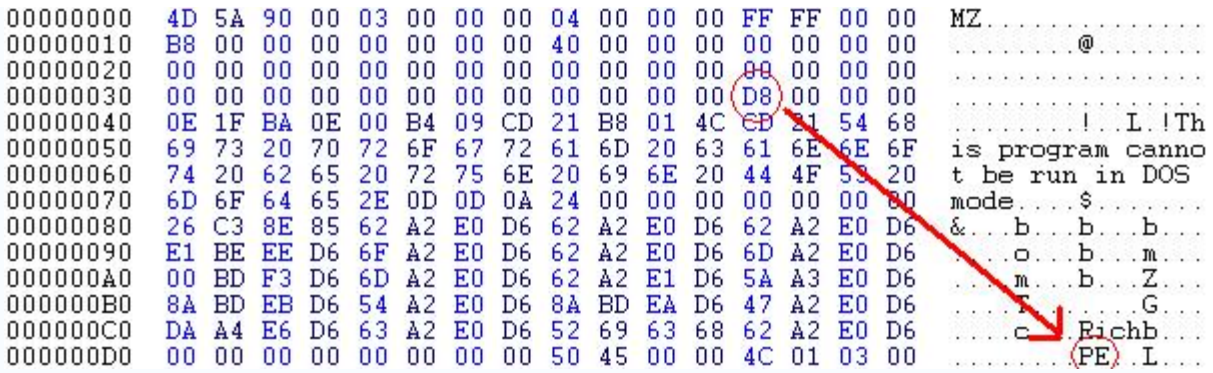
Hex Listing of an MS-DOS file header

What you are looking at is the MS-DOS header of the Win32 PE file. To ensure either a) backwards compatibility, or b) graceful decline of new file types, Microsoft has engineered a series of DOS instructions into the head of each PE file. When a 32-bit Windows file is run in a 16-bit DOS environment, the program will terminate immediately with the error message: “This program cannot be run in DOS mode”. The DOS header is also known by some as the EXE header. Here is the DOS header presented as a C data structure:

```
struct DOS_Header
 {
char signature= “MZ”;
short lastsize;
short nblocks;
short nreloc;
short hdrsize;
short minalloc;
short maxalloc;
void *sp;
short checksum;
void *ip;
short relocpos;
short noverlay;
short reserved1[4];
short oem_id;
short oem_info;
short reserved2[10];
long  e_lfanew;
 }
```

PE Header

At offset 60 from the beginning of the DOS header, is a pointer to the Portable Executable (PE) File header (e_lfanew in MZ structure). DOS will print the error message and terminate, but Windows will follow this pointer to the next batch of information.



Hex Listing of a PE signature, and the pointer to it

The PE header consists only of a File ID signature, with the value “PE\0\0” where each ‘\0’ character is an ASCII NUL character. This signature shows that a) this file is a legitimate PE file, and b) the byte order of the file. Byte order will not be considered in this chapter, and all PE files are assumed to be in “little endian” format. The first big chunch of information lies in the COFF header, directly after the PE signature.

COFF Header

The COFF header is present in both COFF object files (before they are linked) and in PE files where it is known as the “File header”. The COFF header has some information that is useful to an executable, and some information that is more useful to an object file.

Here is the COFF header, presented as a C data structure:

```
struct COFFHeader
{
short Machine;
short NumberOfSections;
long TimeDateStamp;
long PointerToSymbolTable;
long NumberOfSymbols;
short SizeOfOptionalHeader;
short Characteristics;
}
```

Machine

> This field determines what machine the file was compiled for. A hex value of 0x14C (332 in decimal) is the code for an Intel 80386.

NumberOfSections

> The number of sections that are described at the end of the PE headers.

TimeDateStamp

32 bit time at which this header was generated: is used in the process of “Binding”, see below.

SizeOfOptionalHeader

this field shows how long the “PE Optional Header” is that follows the COFF header.

Characteristics

This is a field of bit flags, that show some characteristics of the file.

- 0x02 = Executable file
- 0x200 = file is non-relocatable (addresses are absolute, not RVA).
- 0x2000 = File is a DLL Library, not an EXE.

PE Optional Header

The “PE Optional Header” is not “optional” per se, because it is required in Executable files, but not in COFF object files. The Optional header includes lots and lots of information that can be used to pick apart the file structure, and obtain some useful information about it.

The PE Optional Header occurs directly after the COFF header, and some sources even show the two headers as being part of the same structure. This separates them out for convenience.

Here is the PE Optional Header presented as a C data Structure:

```
struct PEOptHeader
{
short signature; //decimal number 267.
char MajorLinkerVersion;
char MinorLinkerVersion;
long SizeOfCode;
long SizeOfInitializedData;
long SizeOfUninitializedData;
long AddressOfEntryPoint;  //The RVA of the code entry point
long BaseOfCode;
long BaseOfData;
long ImageBase;
long SectionAlignment;
long FileAlignment;
short MajorOSVersion;
short MinorOSVersion;
short MajorImageVersion;
short MinorImageVersion;
short MajorSubsystemVersion;
short MinorSubsystemVersion;
long Reserved;
long SizeOfImage;
long SizeOfHeaders;
long Checksum;
short Subsystem;
short DLLCharacteristics;
long SizeOfStackReserve;
long SizeOfStackCommit;
long SizeOfHeapReserve;
long SizeOfHeapCommit;
```

```
long LoaderFlags;

long NumberOfRvaAndSizes;

   data_directory DataDirectory[16];     //Can have any number of elements, matching the
number in NumberOfRvaAndSizes.
}                                        //However, it is always 16 in PE files.
```

struct data_directory

```
{
long VirtualAddress;
long Size;
}
```

Some of the important pieces of information:

MajorLinkerVersion, MinorLinkerVersion

The version, in x.y format of the linker used to create the PE.

AddressOfEntryPoint

The RVA of the entry point to the executable. Very important to know.

SizeOfCode

Size of the .text (.code) section

SizeOfInitializedData

Size of .data section

BaseOfCode

RVA of the .text section

BaseOfData

RVA of .data section

ImageBase

Preferred location in memory for the module to be based at

Checksum

Checksum of the file, only used to verify validity of modules being loaded into kernel space. The formula used to calculate PE file checksums is proprietary, although Microsoft provides API calls that can calculate the checksum for you.

Subsystem

the Windows subsystem that will be invoked to run the executable

- 1 = native
- 2 = Windows/GUI

- 3 = Windows non-GUI
- 5 = OS/2
- 7 = POSIX

DataDirectory

Possibly the most interesting member of this structure. Provides RVAs and sizes which locate various data structures, which are used for setting up the execution environment of a module. The details of what these structures do exists in other sections of this page, but the most interesting entries in DataDirectory are below:

- IMAGE_DIRECTORY_ENTRY_EXPORT (0) : Location of the export directory
- IMAGE_DIRECTORY_ENTRY_IMPORT (1) : Location of the import directory
- IMAGE_DIRECTORY_ENTRY_RESOURCE (2) : Location of the resource directory
- IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (11) : Location of alternate import-binding directory

Code Sections

The PE Header defines the number of sections in the executable file. Each section definition is 40 bytes in length. Below is an example hex from a program I am writing:

```
2E746578_74000000_00100000_00100000_A8050000 .text
00040000_00000000_00000000_00000000_20000000
2E646174_61000000_00100000_00200000_86050000 .data
000A0000_00000000_00000000_00000000_40000000
2E627373_00000000_00200000_00300000_00000000 .bss
00000000_00000000_00000000_00000000_80000000
```

The structure of the section descriptor is as follows:

```
Offset Length   Purpose
------ -------  ------------------------------------------------------------------
 0x00   8 bytes Section Name - in the above example the names are .text .data .bss
 0x08   4 bytes Size of the section once it is loaded to memory
 0x0C   4 bytes RVA (location) of section once it is loaded to memory
 0x10   4 bytes Physical size of section on disk
 0x14   4 bytes Physical location of section on disk (from start of disk image)
0x18  12 bytes Reserved (usually zero) (used in object formats)
0x24   4 bytes Section flags
```

A PE loader will place the sections of the executable image at the locations specified by these section descriptors (relative to the base address) and usually the alignment is 0x1000, which matches the size of pages on the x86.

Common sections are:

1. .text/.code/CODE/TEXT - Contains executable code (machine instructions)
2. .data/.idata/DATA/IDATA - Contains initialised data

3. .bss/BSS - Contains uninitialised data

Section Flags

The section flags is a 32bit bit field (each bit in the value represents a certain thing). Here are the constants defined in the WINNT.H file for the meaning of the flags:

```
#define IMAGE_SCN_TYPE_NO_PAD                0x00000008  // Reserved.
#define IMAGE_SCN_CNT_CODE                   0x00000020  // Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA       0x00000040  // Section contains initialized
data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA     0x00000080  // Section contains
uninitialized data.
#define IMAGE_SCN_LNK_OTHER                  0x00000100  // Reserved.
#define IMAGE_SCN_LNK_INFO                   0x00000200  // Section contains comments or
some  other type of information.
#define IMAGE_SCN_LNK_REMOVE                 0x00000800  // Section contents will not
become part of image.
#define IMAGE_SCN_LNK_COMDAT                 0x00001000  // Section contents comdat.
#define IMAGE_SCN_NO_DEFER_SPEC_EXC          0x00004000  // Reset speculative exceptions
handling bits in the TLB entries for this section.
#define IMAGE_SCN_GPREL                      0x00008000  // Section content can be
accessed relative to GP
#define IMAGE_SCN_MEM_FARDATA                0x00008000
#define IMAGE_SCN_MEM_PURGEABLE              0x00020000
#define IMAGE_SCN_MEM_16BIT                  0x00020000
#define IMAGE_SCN_MEM_LOCKED                 0x00040000
#define IMAGE_SCN_MEM_PRELOAD                0x00080000
#define IMAGE_SCN_ALIGN_1BYTES               0x00100000  //
#define IMAGE_SCN_ALIGN_2BYTES               0x00200000  //
#define IMAGE_SCN_ALIGN_4BYTES               0x00300000  //
#define IMAGE_SCN_ALIGN_8BYTES               0x00400000  //
#define IMAGE_SCN_ALIGN_16BYTES              0x00500000  // Default alignment if no
others are specified.
#define IMAGE_SCN_ALIGN_32BYTES              0x00600000  //
#define IMAGE_SCN_ALIGN_64BYTES              0x00700000  //
#define IMAGE_SCN_ALIGN_128BYTES             0x00800000  //
#define IMAGE_SCN_ALIGN_256BYTES             0x00900000  //
#define IMAGE_SCN_ALIGN_512BYTES             0x00A00000  //
#define IMAGE_SCN_ALIGN_1024BYTES            0x00B00000  //
#define IMAGE_SCN_ALIGN_2048BYTES            0x00C00000  //
#define IMAGE_SCN_ALIGN_4096BYTES            0x00D00000  //
#define IMAGE_SCN_ALIGN_8192BYTES            0x00E00000  //
#define IMAGE_SCN_ALIGN_MASK                 0x00F00000
#define IMAGE_SCN_LNK_NRELOC_OVFL            0x01000000  // Section contains extended
relocations.
#define IMAGE_SCN_MEM_DISCARDABLE            0x02000000  // Section can be discarded.
#define IMAGE_SCN_MEM_NOT_CACHED             0x04000000  // Section is not cachable.
#define IMAGE_SCN_MEM_NOT_PAGED              0x08000000  // Section is not pageable.
#define IMAGE_SCN_MEM_SHARED                 0x10000000  // Section is shareable.
#define IMAGE_SCN_MEM_EXECUTE                0x20000000  // Section is executable.
#define IMAGE_SCN_MEM_READ                   0x40000000  // Section is readable.
#define IMAGE_SCN_MEM_WRITE                  0x80000000  // Section is writeable.
```

Imports and Exports - Linking to other modules

What is linking?

Whenever a developer writes a program, there are a number of subroutines and functions which are expected to be implemented already, saving the writer the hassle of having to write out more code or work with complex data structures. Instead, the coder need only declare one call to the subroutine, and the linker will decide what happens next.

There are two types of linking that can be used: static and dynamic. Static uses a library of precompiled functions. This precompiled code can be inserted into the final executable to implement a function, saving the programmer a lot of time. In contrast, dynamic linking allows subroutine code to reside in a different file (or *module*), which is loaded at runtime by the operating system. This is also known as a "Dynamically linked library", or DLL. A *library* is a module containing a series of functions or values that can be *exported*. This is different from the term *executable*, which *imports* things from libraries to do what it wants. From here on, "module" means any file of PE format, and a "Library" is any module which exports and imports functions and values.

Dynamically linking has the following benefits:

- It saves disk space, if more than one executable links to the library module
- Allows instant updating of routines, without providing new executables for all applications
- Can save space in memory by mapping the code of a library into more than one process
- Increases abstraction of implementation. The method by which an action is achieved can be modified without the need for reprogramming of applications. This is extremely useful for backward compatibility with operating systems.

This section discusses how this is achieved using the PE file format. An important point to note at this point is that *anything* can be imported or exported between modules, including variables as well as subroutines.

Loading

The downside of dynamically linking modules together, is that at runtime, the software which is initialising an executable must link these modules together. For various reasons, you cannot declare that "The function in this dynamic library will always exist in memory *here*". If that memory address is unavailable or the library is updated, the function will no longer exist there, and the application trying to use it will break. Instead, each module (library or executable) must declare what functions or values it *exports* to other modules, and also what it wishes to *import* from other modules.

As said above, a module cannot declare where in memory it expects a function or value to be. Instead, it declared where *in it's own memory* it expects to find a **pointer** to the value it wishes to import. This permits the module to address any imported value, wherever it turns up in memory.

Exports

Exports are functions and values in one module that have been declared to be shared with other modules. This is done through use of the “Export Directory”, which is used to translate between the name of an export (or “Ordinal”, see below), and a location in memory where the code or data can be found. The start of the export directory is identified by the IMAGE_DIRECTORY_ENTRY_EXPORT entry of the resource directory. All export data must exist in the same section. The directory is headed by the following structure:

```
struct IMAGE_EXPORT_DIRECTORY {
long Characteristics;
long TimeDateStamp;
short MajorVersion;
short MinorVersion;
long Name;
long Base;
long NumberOfFunctions;
long NumberOfNames;
long *AddressOfFunctions;
long *AddressOfNames;
long *AddressOfNameOrdinals;
}
```

The “Characteristics” value is generally unused, TimeDateStamp describes the time the export directory was generated, `MajorVersion` and MinorVersion should describe the version details of the directory, but their nature is undefined. These values have little or no impact on the actual exports themselves. The “Name” value is an RVA to a zero terminated ASCII string, the name of this library name, or module.

Names and Ordinals

Each exported value has both a name and an “ordinal” (a kind of index). The actual exports themselves are described through AddressOfFunctions, which is an RVA to an array of RVA’s, each pointing to a different function or value to be exported. The size of this array is in the value NumberOfFunctions. Each of these functions has an ordinal. The “Base” value is used as the ordinal of the first export, and the next RVA in the array is Base+1, and so forth.

Each entry in the AddressOfFunctions array is identified by a name, found through the RVA AddressOfNames. The data where AddressOfNames points to is an array of RVA’s, of the size NumberOfNames. Each RVA points to a zero terminated ASCII string, each being the name of an export. There is also a second array, pointed to by the RVA in AddressOfNameOrdinals. This is also of size NumberOfNames, but each value is a 16 bit word, each value being an ordinal. These two arrays are parallel and are used to get an export value from AddressOfFunctions. To find an export by name, search the AddressOfNames array for the correct string and then take the corresponding ordinal from the AddressOfNameOrdinals array. This ordinal is then used to get an index to a value in AddressOfFunctions.

Forwarding

As well as being able to export functions and values in a module, the export directory can *forward* an export to another library. This allows more flexibility when re-organising libraries: perhaps some functionality has branched into another module. If so, an export can be forwarded to that library, instead of messy reorganising inside the original module.

Forwarding is achieved by making an RVA in the AddressOfFunctions array point into the section which contains the export directory, something that normal exports should not do. At that location, there should be a zero terminated ASCII string of format “LibraryName.ExportName” for the appropriate place to forward this export to.

Imports

The other half of dynamic linking is importing functions and values into an executable or other module. Before runtime, compilers and linkers do not know where in memory a value that needs to be imported could exist. The import table solves this by creating an array of pointers at runtime, each one pointing to the memory location of an imported value. This array of pointers exists inside of the module at a defined RVA location. In this way, the linker can use addresses inside of the module to access values outside of it.

The Import directory

The start of the import directory is pointed to by both the IMAGE_DIRECTORY_ENTRY_IAT and IMAGE_DIRECTORY_ENTRY_IMPORT entries of the resource directory (the reason for this is uncertain). At that location, there is an array of IMAGE_IMPORT_DESCRIPTORS structures. Each of these identify a library or module that has a value we need to import. The array continues until an entry where all the values are zero. The structure is as follows:

```
struct IMAGE_IMPORT_DESCRIPTOR {
long *OriginalFirstThunk;
long TimeDateStamp;
long ForwarderChain;
long Name;
long *FirstThunk;
}
```

The TimeDateStamp is relevant to the act of “Binding”, see below. The Name value is an RVA to an ASCII string, naming the library to import. ForwarderChain will be explained later. The only thing of interest at this point, are the RVA’s OriginalFirstThunk and FirstThunk. Both these values point to zero terminated arrays of RVA’s, each of which point to a IMAGE_IMPORT_BY_NAMES struct. These two arrays are parralel and point to the same structure, in the same order. The reason for this will become apparent shortly.

Each of these IMAGE_IMPORT_BY_NAMES structs has the following form:

```
struct IMAGE_IMPORT_BY_NAME {
```

```
short Hint;

char Name[1];

}
```

“Name” is an ASCII string of any size that names the value to be imported. This is used when looking up a value in the export directory (see above) through the AddressOfNames array. The “Hint” value is an index into the AddressOfNames array; to save searching for a string, the loader first checks the AddressOfNames entry corresponding to “Hint”.

To summarise: The import table consists of a large array of IMAGE_IMPORT_DESCRIPTOR’s, terminated by an all-zero entry. These descriptors identify a library to import things from. There are then two parralel RVA arrays, each pointing at IMAGE_IMPORT_BY_NAME structures, which identify a specific value to be imported.

Imports at runtime

Using the above import directory, at runtime the loader finds the appropriate modules, loads them into memory, and seeks the correct export. However, to be able to use the export, a pointer to it must be stored somewhere in the importing module’s memory. This is why there are two parralel arrays, OriginalFirstThunk and FirstThunk, identifying IMAGE_IMPORT_BY_NAME structures. Once an imported value has been resolved, the pointer to it is stored in the FirstThunk array. It can then be used at runtime to address imported values.

Bound imports

The PE file format also supports a peculiar feature known as “binding”. The process of loading and resolving import addresses can be time consuming, and in some situations this is to be avoided. If a developer is fairly certain that a library is not going to be updated or changed, then the addresses in memory of imported values will not change each time the application is loaded. So, the import address can be precomputed and stored in the FirstThunk array *before* runtime, allowing the loader to skip resolving the imports - the imports are “bound” to a particular memory location. However, if the versions numbers between modules do not match, or the imported library needs to be relocated, the loader will assume the bound addresses are invalid, and resolve the imports anyway.

The “TimeDateStamp” member of the import directory entry for a module controls binding; if it is set to zero, then the import directory is not bound. If it is non-zero, then it is bound to another module. However, the TimeDateStamp in the import table must match the TimeDateStamp in the bound module’s FileHeader, otherwise the bound values will be discarded by the loader.

Forwarding and binding

Binding can of course be a problem if the bound library / module forwards it's exports to another module. In these cases, the non-forwarded imports can be bound, but the values which get forwarded must be identified so the loader can resolve them. This is done through the ForwarderChain member of the import descriptor. The value of "ForwarderChain" is an index into the FirstThunk and OriginalFirstThunk arrays. The OriginalFirstThunk for that index identifies the IMAGE_IMPORT_BY_NAME structure for a import that needs to be resolved, and the FirstThunk for that index, is the index of another entry that needs to be resolved. This continues until the FirstThunk value is -1, indicating no more forwarded values to import.

Resources

Resource structures

Resources are data items in modules which are difficult to be stored or described using the chosen programming language. This requires a seperate compiler or resource builder, allowing insertion of window forms, icons, and menus. A number of API calls can then be used to retrieve resources from the module. The base of resource data is pointed to by the IMAGE_DIRECTORY_ENTRY_RESOURCE entry of the data directory, and at that location there is an IMAGE_RESOURCE_DIRECTORY structure:

```
struct IMAGE_RESOURCE_DIRECTORY
{
long Characteristics;
long TimeDateStamp;
short MajorVersion;
short MinorVersion;
short NumberOfNamedEntries;
short NumberOfIdEntries;
}
```

Characteristics is unused, and TimeDateStamp is normally the time of creation, although it doesn't matter if it's set or not. MajorVersion and MinorVersion relate to the versioning info of the resources: the fields have no defined values. Immediately following the IMAGE_RESOURCE_DIRECTORY structure is a series of IMAGE_RESOURCE_DIRECTORY_ENTRY s, the number of which are defined by the total of NumberOfNamedEntries and NumberOfIdEntries. The first porition of these entries are for named resources, the latter for ID resources, depending on the values in the IMAGE_RESOURCE_DIRECTORY struct. The actual shape of the resource entry structure is as follows:

```
struct IMAGE_RESOURCE_DIRECTORY_ENTRY
{
long NameId
long *Data
}
```

The NameId value has dual purpose: if the most significant bit (or sign bit) is clear, then the lower 16 bits are an ID number of the resource. Alternatly, if the top bit is set, then the lower 31 bits make up an offset from the start of the resource data to the name string of this particular resource. The Data value is also an offset from the start of the resource data, locating the actual data of the resource.

Layout

The above system of resource directory and entries allows simple storage of resources, by name or ID number. However, this can get very complicated very quickly. Different types of resources, the resources themselves, and instances of resources in other languages can become muddled in just one directory of resources. For this reason, the resource directory has been given a structure to work by, allowing seperation of the different resources.

For this purpose, the “Data” value of resource entries points at another IMAGE_RESOURCE_DIRECTORY structure, forming a tree-diagram like organisation of resources. The first level of resource entries identifies the *type* of resource is being stored: cursors, bitmaps, icons and similar. They use the ID method of identifying the resource entries, of which there are twelve defined values in total. More user defined resource types can be added. Each of these resource entries points at a resource directory, naming the actual resources themselves. These can be of any name or value. These point at yet another resource directory, which uses ID numbers to distinguish languages, allowing different specific resources for systems using a different language. Finally, the entries in the language directory actually provide the offset to the resource data itself. This can be of any format at all.

ELF Files

The **ELF file format** (short for Executable and Linking File) was developed by the Unix System Laboratories to be a successor to previous file formats such as COFF and a.out. In many respects, the ELF format is more powerful and versatile then previous formats, and has widely become the standard on Linux, Solaris, IRIX, and FreeBSD (although the FreeBSD-derived Mac OS X uses the Mach-O format instead). ELF has also been adopted by OpenVMS for Itanium and BeOS for x86.

Historically, Linux has not always used ELF; Red Hat Linux 4 was the first time that distribution used ELF; previous versions had used the a.out format.

File Format

Other Files

MS-DOS COM Files

COM files are loaded into RAM exactly as they appear; no change is made at all from the harddisk image to RAM. This is possible due to the 20-bit memory model of the early x86 line. Two 16-bit registers would have to be set, one dividing the 1MB+64K memory space into 65536 'segments' and one specifying an offset from that. The segment register would be set by DOS and the COM file would be expected to respect this setting and not ever change the segment registers. The offset registers, however, were free game and served (for COM files) the same purpose as a modern 32-bit register. The downside was that the offset registers were only 16-bit and, therefore, since COM files could not change the segment registers, COM files were limited to using 64K of RAM. The good thing about this approach, however, was that no extra work was needed by DOS to load and run a COM file: just load the file, set the segment register, and jmp to it. (The programs could perform 'near' jumps by just giving an offset to jump too.)

COM files are loaded into RAM at offset $100. The space before that would be used for passing data to and from DOS (for example, the contents of the command line used to invoke the program).

Note that COM files, by definition, cannot be 32-bit. Windows provides support for COM files via a special CPU mode.

a.out Files

Dynamic Libraries

Be they DLLs, or relocatable ELF modules, nearly every robust operating system has some mechanism for dynamic linking. This chapter will discuss some of the common "flavors" of the dynamic code library, but won't go too deep into specifics (because the specifics should have already been covered in the PE File and ELF File sections).

Windows DLL Files

Windows DLL files are a brand of PE file with a few key differences:

- A .DLL file extension
- A DLLMain() entry point, instead of a WinMain() or main().
- The DLL flag set in the PE header.

DLLs may be loaded in one of two ways, a) at load-time, or b) by calling the LoadModule() Win32 API function.

Function Exports

Functions are exported from a DLL file by using the following syntax:

```
__declspec(dllexport) void MyFunction() ...
```

The "__declspec" keyword here is not a C language standard, but is implemented by many compilers to set extendable, compiler-specific options for functions and variables. Microsoft C Compiler and GCC versions that run on windows allow for the __declspec keyword, and the dllexport property.

Functions may also be exported from regular .exe files, and .exe files with exported functions may be called dynamically in a similar manner to .dll files. This is a rare occurance, however.

Identifying DLL Exports

There are several ways to determine which functions are exported by a DLL. The method that this book will use (often implicitly) is to use **dumpbin** in the following manner:

```
dumpbin /EXPORTS <dll file>
```

This will post a list of the function exports, along with their ordinal and RVA to the console.

Function Imports

In a similar manner to function exports, a program may import a function from an external DLL file. The dll file will load into the process memory when the program is started, and the function will be used like a local function. DLL imports need to be prototyped in the following manner, for the compiler and linker to recognize that the function is coming from an external library:

```
__declspec(dllimport) void MyFunction();
```

Identifying DLL Imports

If is often useful to determine which functions are imported from external libraries when examining a program. To list import files to the console, use **dumpbin** in the following manner:

```
dumpbin /IMPORTS <dll file>
```

Relocatable ELF Files

File Formats

This section will talk about reverse-engineering proprietary file formats. This chapter might eventually even include a discussion on reverse-engineering file systems. Many software developers need to reverse engineer a proprietary file format, especially for the purposes of interoperability. For instance, the Open Office project needs to reverse engineer the Microsoft Office file formats on a yearly basis. The chapters in this section will talk about to how understand a proprietary file format.

Note on the Section

This entire section is in need of some help and contributions. If you know anything about this field of study, please help and contribute.

file header

Most file formats begin with a “header”, a few bytes that describe the file type and version. Because there are several incompatible file formats with the same extension (for example, “.doc” and “.cod”), the header gives a program enough additional information to see if this file is one of the formats that program can handle.

Many programmers package their data in some sort of “container format” before writing it out to disk. If they use the standard zlib to hold their data in compressed form, the file will begin with the 2 bytes 0x1f 0x8b (in decimal, 31 139).

Section 9: Anti-Reversing

Anti-Reversing

This section is all about techniques for a programmer to foil reverse-engineering attempts on their software. Anti-Reversing techniques often need to be targeted to a single reversing tool, so we will discuss them all in detail. The following list is a proposed list of topics that this chapter will cover eventually.

Obfuscators

Obfuscators

There are a number of tools on the market that will automate the process of code obfuscation. These products will use a number of transformations to turn a code snippet into a less-readable

form, although it will not affect the program flow itself (although the transformations may increase code size or execution time).

Code Transformations

Code Transformations

We can best demonstrate this technique by example. Let’s say that we have 2 functions, FunctionA and FunctionB. Both of these two functions are comprised of 3 separate parts, which are performed in order. We can break this down as such:

```
FunctionA()
{
FuncAPart1();
FuncAPart2();
FuncAPart3();
}
FunctionB()
{
FuncBPart1();
FuncBPart2();
FuncBPart3();
}
```

And we have our main program, that executes the two functions:

```
main()
{
FunctionA();
FunctionB();
}
```

Now, we can rearrange these snippets to a form that is much more complicated (in assembly):

```
main:
jmp FAP1
FBP3: call FuncBPart3
jmp end
FBP1: call FuncBPart1
jmp FBP2
FAP2: call FuncAPart2
jmp FAP3
FBP2: call FuncBPart2
jmp FBP3
FAP1: call FuncAPart1
jmp FAP2
FAP3: call FuncAPart3
jmp FBP1
end:
```

As you can see, this is much harder to read, although it perfectly preserves the program flow of the original code (don't believe me? trace it yourself). This code is much harder for a human to read, although it isn't hard at all for an automated debugging tool (such as IDA Pro) to read.

Opaque Predicates

Opaque Predicate

An **Opaque Predicate** is a line (or lines) of code in a program that don't do anything, but that *look like they do something.* This is opposed to a transparent predicate that doesnt do anything and looks useless. A program filled with opaque predicates will take more time to decipher, because the reverser will take more time reading through useless, distraction code.

Code Encryption-Decryption

Code Encryption

Code can be encrypted, just like any other type of data, except that code can also work to encrypt and decrypt *itself.* This page will talk about the process of encrypting and decrypting code at runtime.

Basic Rules

There are some basic rules to follow on this topic:

1. Don't decrypt the entire program at once. This is important because if the program is ever 100% decrypted, a hacker can dump the memory, and obtain a decrypted listing of the program. Different parts of the program should be decrypted and re-encrypted one at a time, as they are being used, for maximum security.
2. Calculate the decryption key at runtime. This prevents hackers from getting the key from the code, running an external decrypter, and obtaining the decrypted code. Throw it away immediately after using it to make it more difficult for the key to be picked up out of memory.
3. Use Large Keys. 32bit keys can be easily decrypted using a moderately fast computer, a quick algorithm, and a brute-force approach. 64bit keys or longer should be used to ensure security.

Issues

A skilled hacker will be able to get your decryption key anyway, by setting a hardware breakpoint at the spot where you finish calculating it.

Code Hashing

Simple Checksums

Most BIOSes[10] and Embedded Systems calculate a “checksum” of the software before executing it (or at least, before executing anything past the “checksum” code). Then if the calculated checksum fails to match the “correct” checksum, the BIOS refuses to execute the “corrupted” software. For example, the original IBM PC calculated the checksum in the ROM area for each ISA plug-in card, and only executed that ROM if the checksum ended in “00”.

The checksum assured the programmer that the program really was all there. Then if something weird happened later on, he knew it was an actual bug in the code—he couldn’t blame it problems with burning and installing the ROMs, or spontaneous floppy data failures, or corruption while downloading software over phone lines, etc.

If a reverse engineer experiments with modifying a few bytes of the software, then these checksums will fail, and the software will refuse to run. But many checksums are simple enough that it’s easy to modify one or two non-critical bytes to force the sum to equal the “correct” checksum. Hashes, though, are a different story.

Hashes

People familiar with the concept of finding hashes of information, will find this chapter more interesting than those who don’t. Let’s say we use a known hashing algorithm to find the given hash value of our program’s code. Now, we can have the program recalculate the hash value at run-time, and compare this value to the known value. If the two numbers don’t match up, then we know that a hacker has patched the code. At this point, we can terminate the program with a warning: “Don’t hack this program”.

A good example of code hashing techniques is the .NET platform. .NET allows a programmer to sign a hash value or a “signature” to a compiled .NET module. If the code has been edited or patched in any way, the program will not run.

Detecting Debuggers

Detecting Debuggers

It may come as a surprise that a running program can actually detect the presence of an attached user-mode debugger. Also, there are methods available to detect kernel-mode debuggers, although the methods used depend in large part on which debugger is trying to be detected.

IsDebuggerPresent API

The Win32 API contains a function called “IsDebuggerPresent”, which will return a boolean true if the program is being debugged. The following code snippet will detail a general usage of this function:

```
if(IsDebuggerPresent())
{
TerminateProcess(GetCurrentProcess(), 1);
}
```

Of course, it is easy to spot uses of the IsDebuggerPresent() function in the disassembled code, and a skilled reverser will simply patch the code to remove this line.

Timeouts

Debuggers can put break points in the code, and can therefore stop program execution. A program can detect this, by monitoring the system clock. If too much time has elapsed between instructions, it can be determined that the program is being stopped and analyzed (although this is not always the case). If a program is taking too much time, the program can terminate.

Detecting SoftICE

SoftICE is a local kernel debugger, and as such, it can’t be detected as easily as a user-mode debugger can be. The IsDebuggerPresent API function will not detect the presence of SoftICE.

To detect SoftICE, there are a number of techniques that can be used:

1. Search for the SoftICE install directory. If SoftICE is installed, the user is probably a hacker or a reverser.
2. Detect the presence of **int 1**. SoftICE uses interrupt 1 to debug, so if interrupt 1 is installed, SoftICE is running.

Detecting OllyDbg

OllyDbg is a popular 32-bit usermode debugger. Unfortunately, the last few releases, including the latest version (v1.10) contain a vulnerability in the handling of the Win32 API function OutputDebugString(). [11] A programmer trying to prevent his program from being debugged by OllyDbg could exploit this vulnerability in order to make the debugger crash. The author has never released a fix, however there are unofficial versions and plugins available to protect OllyDbg from being exploited using this vulnerability.

List of Examples

Examples

This section will present a series of Examples and larger Case studies that will draw on information from previous sections. Examples and Case studies do not need to be “real world” code, but may be fabricated for the purposes of this book. when possible, the original High-Level Source should be included as an addendum. The focus here will be in real-world examples, although some examples lend themselves much more easily (and much more legally) to ficticious examples.

- Calling Conventions Examples
- Variables Examples
- Branches Examples
- Loops Examples
- Code Optimization Examples

Examining Proprietary File Formats

This section will contain a series of examples and case studies.

Malicious Code and Security

This section will contain a series of examples and case studies

Identifying “Hidden” APIs

This section will contain a series of examples and case studies

Using Undocumented 3rd Party Libraries

This section will contain a series of examples and case studies

Real-World Examples

This book will also contain a listing of current reverse engineering projects that are a) legal, and b) useful in teaching about reverse engineering. We may also include pages on these projects to talk about how they are going.

CVS One-Time-Use Video Camera

> There is a large effort in the reversing community to reverse engineer the CVS one-time-use video camera.

CVS One-Time-Use Camera

Functions and Stack Frames

To allow for many unknowns in the execution environment, functions are frequently set up with a "**stack frame**" to allow access to both function parameters, and automatic function variables. The idea behind a stack frame is that each subroutine can act independently of its location on the stack, and each subroutine can act as if it is the top of the stack.

Standard Entry Sequence

For many compilers, the standard function entry sequence is the following piece of code *:

```
push ebp
mov ebp, esp
sub esp, X
```

Where *X* is the total size, in bytes, of all *automatic* variables used in the function. For example, here is a C function, and the resulting assembly instructions:

```
void MyFunction()
{
int a, b, c;
  ...
push ebp      ; save value of ebp
mov ebp, esp  ; ebp points to the top of the stack
sub esp, 12   ; space allocated on the stack for local variables
```

In this manner, local variables can be accessed by referencing esp. The following C code will demonstrate how local variables are referenced in assembly code:

```
 ...
x = 10;
y = 5;
z = 2;
 ...
mov [esp + 0], 10  ; [esp + 0] is the location of variable a
 mov [esp + 4], 5   ; location of b
 mov [esp + 8], 2   ; location of c
```

This all seems well and good, but what is the purpose of **ebp** in this setup? Why save the old value of ebp, and why point ebp to the top of the stack, only to change the value of esp with the next instruction? The answer is *function parameters*.

Consider the following C function declaration:

```
void MyFunction2(int x, int y, int z)
```

```
{

   ...

}
```

This function produces the following assembly code:

```
push ebp
mov ebp, esp
sub esp, 0     ; no local variables, most compilers will omit this line
```

Which is exactly as one would expect. So, what exactly does **ebp** do, and where are the function parameters stored? The answer is found when we call the function.

Consider the following C function call:

```
MyFunction2(10, 5, 2);
```

This will create the following assembly code (using a Right-to-Left calling convention called CDECL, explained later):

```
push 2
push 5
push 10
call _MyFunction2
```

It turns out that the function arguments are all passed on the stack! Therefore, when we move the current value of the stack pointer (**esp**) into **ebp**, we are pointing ebp directly at the function arguments.

Note: It must be remembered that the **call** x86 instruction basically performs the following actions:

```
push eip + 2
jmp _MyFunction2
```

This means that first the return address and then the old value of **ebp** are put on the stack. Therefore [ebp] points to the location of the old value of ebp, [ebp + 4] points to the return address, and [ebp + 8] points to the first function argument. Here is a *very crude* drawing of the stack at this point:

```
Stack
|    | ...
|    | [ebp + 8] (First argument)
|    | [ebp + 4] (return address)

|    | [ebp + 0] (old ebp value)

|....| ...
|    | [esp + 0] (first variable
|    |
```

- *ASM code here is in MASM syntax*

Standard Exit Sequence

The Standard Exit Sequence must undo the things that the Standard Entry Sequence does. To this effect, the Standard Exit Sequence must perform the following tasks, in the following order:

1. Remove space for local variables, by reverting **esp** to its old value.
2. Restore the old value of **ebp** to its old value, which is on top of the stack.
3. Return to the calling function with a *ret* command.

As an example, the following C code:

```
void MyFunction3(int x, int y, int z)
{
int a, int b, int c;
  ...
return;
}
```

Will create the following assembly code*:

```
push ebp
mov ebp, esp
sub esp, 12 ; sizeof(a) + sizeof(b) + sizeof(c)
;z = [ebp + 8] y = [ebp + 12] x = [ebp + 16]
;a = [esp + 0] b = [esp + 4]  c = [esp + 8]
mov esp, ebp
pop ebp
ret
```

*ASM code in this example is in MASM syntax

Non-Standard Stack Frames

Frequently, reversers will come across a subroutine that doesn't set up a standard stack frame. Here are some things to consider when looking at a subroutine that does not start with a standard sequence:

Using Uninitialized Registers

When a subroutine starts using data in an *uninitialized* register, that means that the subroutine expects external functions to put data into that register before it gets called. Some calling conventions pass arguments in registers, but sometimes a compiler will not use a standard calling convention.

"static" functions

In C, functions may optionally be declared with the **static** keyword, as such:

```
static void MyFunction4();
```

The **static** keyword causes a function to have only local scope, meaning it may not be accessed by any external functions (it is strictly internal to the given code file). When an optimizing compiler sees a static function, it "knows" that external functions cannot possibly interface with the static function (the compiler controls all access to the function), so the compiler doesn't bother making it standard.

Local Static Variables

Local static variables cannot be created on the stack, since the value of the variable is preserved across function calls. Check out the Variables chapter for a discussion of how static variables are implemented.

Stack Frames Questions

Question 1

Given the following disassembled function (in MASM syntax), how many 4-byte parameters does this function receive? How many variables are created on the stack? What does this function do?

```
push ebp
mov ebp, esp
sub esp, 4
mov eax, [ebp + 8]
mul 2
mov [esp + 0], eax
mov eax, [ebp + 12]
mov edx, [esp + 0]
```

```
add eax, edx
mov esp, ebp
pop ebp
ret
```

Question 2

Does the following function follow the Standard Entry and Exit Sequences? if not, where does it differ?

```
:_Question2
call _SubQuestion2
mul 2
ret
```

Stack Frames Answers

Answer 1

The function above takes 2 4-byte parameters, accessed by offsets +8 and +12 from ebp. The function also has 1 variable created on the stack, accessed by offset +0 from esp. The function is nearly identical to this C code:

```
int Question1(int x, int y)
{
int z;
z = x * 2;
return y + z;
}
```

Answer 2

The function does not follow the standard entry sequence, because it doesnt set up a proper stack frame with ebp and esp. The function basically performs the following C instructions:

```
int Question2()
{
return SubQuestion2() * 2;
}
```

Although an optimizing compiler has chosen to take a few shortcuts.

Calling Conventions

Calling conventions are a standardized method for functions to be implemented and called by the machine.

A calling convention specifies the method that a compiler sets up to access a subroutine. In theory, code from any compiler can be interfaced together, so long as the functions all have the same calling conventions. In practice however, this is not always the case.

Notes on Terminology

There are a few terms that we are going to be using in this chapter, which are mostly common sense, but which are worthy of stating directly:

Passing arguments

"passing arguments" is a way of saying that we are putting our arguments in the place where our function will look for them. Arguments are passed before the *call* instruction is executed.

Right-to-Left and Left-to-Right

These describe the manner that arguments are passed to the subroutine, in terms of the High-level code. For instance, the following C function call:

```
MyFunction1(a, b);
```

will generate the following code if passed Left-to-Right:

```
push a
push b
call _MyFunction
```

and will generate the following code if passed Right-to-Left:

```
push b
push a
call _MyFunction
```

Return value

Some functions return a value, and that value must be received reliably by the function's caller. The called function places it's return value in a place where the

calling function can get it when execution returns. Return values must be handled before the called function executes the *ret* instruction.

Cleaning the stack

When arguments are pushed onto the stack, eventually they must be popped back off again. Whichever function is responsible for cleaning the stack must reset the stack pointer to eliminate the passed arguments.

Calling function

The “parent” function that calls the subroutine. Execution resumes in the calling function directly after the subroutine call, unless the program terminates inside the subroutine.

Called function

The “Child” function that gets called by the “parent.”

Name Decoration

When C code is translated to assembly code, the compiler will often “decorate” the function name by adding extra information that the linker will use to find and link to the correct functions. For most calling conventions, the decoration is very simple (often only an extra symbol or two to denote the calling convention), but in some extreme cases (notably C++ “thiscall” convention), the names are “mangled” severely.

Standard C Calling Conventions

The C language, by default, uses the CDECL calling convention, but most compilers allow the programmer to specify another convention via a specifier keyword. These keywords **are not** part of the ISO-ANSI C standard, so you should always check with your compiler documentaion about implementation specifics.

if a calling convention other then CDECL is to be used, or if CDECL is not default for your compiler, and you want to manually use it, you must specify the calling convention keyword in the function declaration itself, and in any prototypes for the function. This is important because both the calling function and the called function need to know the calling convention.

CDECL

In the CDECL calling convention the following holds:

- Arguments are passed on the stack in Right-to-Left order, and return values are passed in eax.
- The *Calling* function cleans the stack. This allows CDECL functions to have *variable-length argument lists*. For this reason the number of arguments is not appended to the name of the function by the compiler, and the assembler and the linker are therefore unable to determine if an incorrect number of arguments is used.

Variable-length argument lists may be considered briefly in a later chapter (or as an eventual addendum to this chapter).

Consider the following C instructions:

```
_cdecl int MyFunction1(int a, int b)
{
return a + b;
}
```

and the following function call:

```
x = MyFunction1(2, 3);
```

These would produce the following assembly listings, respectively:

```
:_MyFunction1
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add eax, edx
pop ebp
ret
push 3
push 2
call _MyFunction1
add esp, 8
```

When translated to assembly code, CDECL functions are almost always prepended with an underscore (that's why all previous examples have used "_" in the assembly code).

STDCALL

STDCALL, also known as "WINAPI" (and a few other names, depending on where you are reading it) is used almost exclusively by Microsoft as the standard calling convention for the Win32 API. Since STDCALL is strictly defined by Microsoft, all compilers that implement it do it the same way.

- STDCALL passes arguments right-to-left, and returns the value in eax. (The Microsoft documentation erroneously claims that arguments are passed left-to-right, but this is not the case: see the Calling Conventions Example page for details.)
- The called function cleans the stack, unlike CDECL. This means that STDCALL doesn't allow variable-length argument lists.

Consider the following C function:

```
_stdcall int MyFunction2(int a, int b)
{
return a + b;
}
```

and the calling instruction:

```
x = MyFunction2(2, 3);
```

These will produce the following respective assembly code fragments:

```
:_MyFunction@8
push ebp
mov ebp, esp
mov eax, [ebp + 8]
mov edx, [ebp + 12]
add edx
pop ebp
ret 8
push 3
push 2
call _MyFunction@8
```

There are a few important points to note here:

1. In the function body, the *ret* instruction has an (optional) argument that indicates how many bytes to pop off the stack when the function returns.
2. STDCALL functions are name-decorated with a leading underscore, followed by an @, and then the number (in bytes) of arguments passed on the stack. This number will always be a multiple of 4, on a 32-bit aligned machine.

FASTCALL

The FASTCALL calling convention is not completely standard across all compilers, so it should be used with caution. In FASTCALL, the first 2 or 3 32-bit (or smaller) arguments are passed in registers, with the most commonly used registers being edx, eax, and ecx. Additional arguments, or arguments larger then 4-bytes are passed on the stack, often in Right-to-Left order (similar to CDECL). The calling function most frequently is responsible for cleaning the stack, if needed.

For specifics of how individual compilers implement FASTCALL, see the Calling Conventions Examples page.

Because of the ambiguities, it is recommended that FASTCALL be used only in situations with 1, 2, or 3 32-bit arguments, where speed is essential.

The following C function:

```
_fastcall int MyFunction3(int a, int b)
{
return a + b;
}
```

and the following C function call:

```
x = MyFunction3(2, 3);
```

Will produce the following assembly code fragments for the called, and the calling functions, respectively:

```
:@MyFunction3@8
push ebp
mov ebp, esp ;many compilers create a stack frame even if it isnt used
add eax, edx ;a is in eax, b is in edx
pop ebp
ret
;the calling function
mov eax, 2
mov edx, 3
call @MyFunction3@8
```

The name Decoration for FASTCALL prepends an @ to the function name, and follows the function name with @x, where x is the number (in bytes) of arguments passed to the function.

Many compilers still produce a stack frame for FASTCALL functions, especially in situations where the FASTCALL function itself calls another subroutine. However, if a FASTCALL function doesn’t need a stack frame, optimizing compilers are free to omit it.

C++ Calling Convention

C++ requires that non-static methods of a class be called by an instance of the class. Therefore it uses its own standard calling convention to ensure that pointers to the object are passed to the function: **THISCALL**.

THISCALL

In THISCALL, the pointer to the class object is passed in ecx, the arguments are passed Right-to-Left on the stack, and the return value is passed in eax.

For instance, the following C++ instruction:

```
MyObj.MyMethod(a, b, c);
```

Would form the following asm code:

```
mov ecx, MyObj
push c
push b
push a
call _MyMethod
```

At least, it *would* look like the assembly code above if it weren't for a nasty practice.

Because of the complexities inherent in function overloading, C++ functions are heavily name-decorated to the point that people often refer to the process as "Name Mangling." Unfortunately C++ compilers are free to do the name-mangling differently since the standard does not enforce a convention, and anyway other issues such as exception handling are certainly not standard anyway.

Since every compiler does the name-mangling differently, this book will not spend too much time discussing the specifics of the algorithm.

Here are a few general remarks about THISCALL name-mangled functions:

- They are recognizable on sight because of their complexity when compared to CDECL, FASTCALL, and STDCALL function name decorations
- They sometimes include the name of that function's class.
- They almost always include the number and type of the arguments, so that overloaded functions can be differentiated by the arguments passed to it.

Here is an example of a C++ class and function declaration:

```
class MyClass {
MyFunction(int a);
}

MyClass::MyFunction(2)
```

and here is the resultant mangled name:

```
?MyFunction@MyClass@@QAEHH@Z
```

Other Language Calling Conventions

Pascal

Fortran

Ada

Calling Conventions Examples

This page will show a series of examples on how different calling conventions are implemented on different compilers. For the main discussion of calling conventions, see Calling Conventions.

Here is a table showing how arguments are passed in different calling conventions:

Compiler	CDECL	FASTCALL	STDCALL
Microsoft C Compiler	right-to-left	arg1 → ecx arg2 → edx the rest right-to-left	right-to-left
GNU GCC	right-to-left	arg1 → ecx arg2 → edx the rest right-to-left	right-to-left

Microsoft C Compiler

Here is a simple function in C:

```
int MyFunction(int x, int y)
{
return (x * 2) + (y * 3);
}
```

using cl.exe, we are going to generate 3 separate listings for MyFunction, one with CDECL, one with FASTCALL, and one with STDCALL calling conventions. On the commandline, there are several switches that you can use to force the compiler to change the default:

- /Gd : The default calling convention is CDECL
- /Gr : The default calling convention is FASTCALL
- /Gz : The default calling convention is STDCALL

Using these commandline options, here are the listings:

CDECL

```
int MyFunction(int x, int y)
{
return (x * 2) + (y * 3);
}
```

becomes:

```
PUBLIC     _MyFunction
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 4
_y$ = 12                                                        ; size = 4
_MyFunction     PROC NEAR
; Line 4
        push    ebp
        mov     ebp, esp
; Line 5
        mov     eax, DWORD PTR _y$[ebp]
        imul    eax, 3
        mov     ecx, DWORD PTR _x$[ebp]
        lea     eax, DWORD PTR [eax+ecx*2]
; Line 6
        pop     ebp
        ret     0
_MyFunction     ENDP
_TEXT   ENDS
END
```

As you can clearly see, parameter y was pushed first, because it has a higher offset from ebp then x does. Both x and y are accessed as offsets from ebp, so we know they are located on the stack. For that matter, the function sets up a standard stack frame as well. The function does not clean it's own stack, as you can see from the "ret 0" instruction at the end. It is therefore the callers duty to clean the stack after the function call.

As a point of interest, notice how **lea** is used in this function to simultaneously perform the multiplication (ecx * 2), and the addition of that quantity to eax. Unintuitive instructions like this will be explored further in the chapter on Unintuitive Instructions.

FASTCALL

```
int MyFunction(int x, int y)
{
return (x * 2) + (y * 3);
}
```

becomes:

```
PUBLIC     @MyFunction@8
_TEXT   SEGMENT
_y$ = -8                                                ; size = 4
_x$ = -4                                                ; size = 4
@MyFunction@8 PROC NEAR
; _x$ = ecx
; _y$ = edx
; Line 4
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR _y$[ebp], edx
        mov     DWORD PTR _x$[ebp], ecx
; Line 5
        mov     eax, DWORD PTR _y$[ebp]
        imul    eax, 3
        mov     ecx, DWORD PTR _x$[ebp]
        lea     eax, DWORD PTR [eax+ecx*2]
; Line 6
        mov     esp, ebp
        pop     ebp
        ret     0
@MyFunction@8 ENDP
_TEXT   ENDS
END
```

This listing is very interesting. I want the reader to keep one important point in mind before I start talking about this function: This function was compiled with *optimizations turned off*. With that point in mind, let's examine it a little bit. First thing we notice is that on line 4, a standard stack frame is set up, and then *ebp is decremented by 8*. Why does it do this? The function might not receive the parameters on the stack, but the cl.exe code generation back-end is expecting the parameters to be on the stack anyway! This means that space needs to be allocated on the stack, and the parameters need to be moved out of ecx and edx, and moved onto the stack. This is made even more rediculous by the fact that parameter x is moved out of ecx in the beginning of the function, and is moved *back into ecx* on line 5. Hopefully the optimizer would catch this nonsense if the optimizer was turned on.

It is difficult to determine which parameter is passed "first" because they are not put in sequential memory addresses like they would be on the stack. However, the Microsoft documentation claims that cl.exe passes fastcall parameters from left-to-right. To prove this point, let's examine a simple little function with only one parameter, to see which register it is passed in:

```
int FastTest(int z)
{
```

```
return z * 2;

}
```

And cl.exe compiles this listing:

```
PUBLIC     @FastTest@4
_TEXT   SEGMENT
_z$ = -4                                                ; size = 4
@FastTest@4 PROC NEAR
; _z$ = ecx
; Line 2
        push    ebp
        mov     ebp, esp
        push    ecx
        mov     DWORD PTR _z$[ebp], ecx
; Line 3
        mov     eax, DWORD PTR _z$[ebp]
        shl     eax, 1
; Line 4
        mov     esp, ebp
        pop     ebp
        ret     0
@FastTest@4 ENDP
_TEXT   ENDS
END
```

So it turns out that the first parameter passed is passed in ecx. We notice in our function above that the first parameter (x) was passed in ecx as well. Therefore, parameters really are passed left-to-right, unlike in CDECL.

Notice 2 more details:

- The name-decoration scheme of the function: @MyFunction@8.
- The “ret 0” function seems to show that the caller cleans the stack, but in this case, there is nothing on the stack to clean. It is unclear who will clean the stack, from this listing. If we take a look at yet one more mini-example:

```
int FastTest(int x, int y, int z, int a, int b, int c)
{
return x * y * z * a * b * c;
}
```

and the corresponding listing:

```
PUBLIC     @FastTest@24
_TEXT   SEGMENT
_y$ = -8                                                ; size = 4
_x$ = -4                                                ; size = 4
_z$ = 8                                             ; size = 4
```

```
_a$ = 12                                                ; size = 4
_b$ = 16                                                ; size = 4
_c$ = 20                                                ; size = 4
@FastTest@24 PROC NEAR

; _x$ = ecx

; _y$ = edx
; Line 2
        push    ebp
        mov     ebp, esp
        sub     esp, 8
        mov     DWORD PTR _y$[ebp], edx
        mov     DWORD PTR _x$[ebp], ecx
; Line 3
        mov     eax, DWORD PTR _x$[ebp]
        imul    eax, DWORD PTR _y$[ebp]
        imul    eax, DWORD PTR _z$[ebp]
        imul    eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        imul    eax, DWORD PTR _c$[ebp]
; Line 4
        mov     esp, ebp
        pop     ebp
        ret     16                                      ; 00000010H
```

We can clearly see that in this case, the callee is cleaning the stack, which we can safely assume will happen every time. An important point to notice about this function is that only the first 2 parameters are passed in registers. The first of which is passed in ecx, and the second of which is passed in edx. All the remaining arguments are clearly passed on the stack, in **right-to-left order**. It seems that the first 2 arguments are passed left-to-right, but all the remaining arguments are passed right-to-left.

STDCALL

```
int MyFunction(int x, int y)
{
return (x * 2) + (y * 3);
}
```

becomes:

```
PUBLIC     _MyFunction@8
_TEXT   SEGMENT
_x$ = 8                                             ; size = 4
_y$ = 12                                                ; size = 4
_MyFunction@8 PROC NEAR
; Line 4
        push    ebp
        mov     ebp, esp
; Line 5
        mov     eax, DWORD PTR _y$[ebp]
        imul    eax, 3
        mov     ecx, DWORD PTR _x$[ebp]
```

```
        lea     eax, DWORD PTR [eax+ecx*2]
; Line 6
        pop     ebp
        ret     8
_MyFunction@8 ENDP
_TEXT   ENDS
END
```

Notice that y is a higher offset from ebp, which indicates that these arguments are passed on the stack from left-to-right, instead of right-to-left as the Microsoft documentation claims. The proof is in the pudding, it would seem. The STDCALL listing is almost identical to the CDECL listing except for the last instruction, which says “ret 8”. This function is clearly cleaning it’s own stack. Notice the name-decoration scheme, with an underscore in front, and an “@8” on the end, to denote how many bytes of arguments are passed. Lets do an example with more parameters:

```
int STDCALLTest(int x, int y, int z, int a, int b, int c)
{
return x * y * z * a * b * c;
}
```

Let’s take a look at how this function gets translated into assembly by cl.exe:

```
PUBLIC     _STDCALLTest@24
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 4
_y$ = 12                                                        ; size = 4
_z$ = 16                                                        ; size = 4
_a$ = 20                                                        ; size = 4
_b$ = 24                                                        ; size = 4
_c$ = 28                                                        ; size = 4
_STDCALLTest@24 PROC NEAR
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        mov     eax, DWORD PTR _x$[ebp]
        imul    eax, DWORD PTR _y$[ebp]
        imul    eax, DWORD PTR _z$[ebp]
        imul    eax, DWORD PTR _a$[ebp]
        imul    eax, DWORD PTR _b$[ebp]
        imul    eax, DWORD PTR _c$[ebp]
; Line 4
        pop     ebp
        ret     24                                              ; 00000018H
_STDCALLTest@24 ENDP
_TEXT   ENDS
END
```

Notice the name decoration, and how there is now “@24” appended to the name, to signify the fact that there are 24 bytes worth of parameters. Notice also how x has the lowest offset, and how c has the highest offset, indicating that c (the right-most parameter) was passed first, and that x

(the left-most parameter) was passed last. Therefore it's clearly a right-to-left passing order. The "ret 24" statement at the end cleans 24 bytes off the stack, exactly like one would expect.

GNU C Compiler: GCC

We will be using 2 example C functions to demonstrate how GCC implements calling conventions:

```
int MyFunction1(int x, int y)
{
return (x * 2) + (y * 3);
}
```

and

```
int MyFunction2(int x, int y, int z, int a, int b, int c)
{
return x * y * (z + 1) * (a + 2) * (b + 3) * (c + 4);
}
```

GCC does not have commandline arguments to force the default calling convention to change from CDECL (for C), so they will be manually defined in the text with the directives: __cdecl, __fastcall, and __stdcall.

CDECL

The first function (MyFunction1) provides the following assembly listing:

```
_MyFunction1:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        leal    (%eax,%eax), %ecx
        movl    12(%ebp), %edx
        movl    %edx, %eax
        addl    %eax, %eax
        addl    %edx, %eax
        leal    (%eax,%ecx), %eax
        popl    %ebp
ret
```

First of all, we can see the name-decoration is the same as in cl.exe. We can also see that the ret instruction doesnt have an argument, so the calling function is cleaning the stack. However, since GCC doesnt provide us with the variable names in the listing, we have to deduce which parameters are which. After the stack frame is set up, the first instruction of the function is "movl 8(%ebp), %eax". One we remember (or learn for the first time) that GAS instructions have the general form:

```
instruction src, dest
```

We realize that the value at offset +8 from ebp (the last parameter pushed on the stack) is moved into eax. The leal instruction is a little more difficult to decipher, especially if we don't have any experiance with GAS instructions. The form "leal(reg1,reg2), dest" adds the values in the parenthesis together, and stores the value in *dest*. Translated into Intel syntax, we get the instruction:

```
lea ecx, [eax + eax]
```

Which is clearly the same as a multiplication by 2. The first value accessed must then have been the last value passed, which would seem to indicate that values are passed right-to-left here. To prove this, we will look at the next section of the listing:

```
movl       12(%ebp), %edx
movl    %edx, %eax
addl    %eax, %eax
addl    %edx, %eax
leal    (%eax,%ecx), %eax
```

the value at offset +12 from ebp is moved into edx. edx is then moved into eax. eax is then added to itselt (eax * 2), and then is added back to edx (edx + eax). remember though that eax = 2 * edx, so the result is edx * 3. This then is clearly the y parameter, which is furthest on the stack, and was therefore the first pushed. CDECL then on GCC is implemented by passing arguments on the stack in right-to-left order, same as cl.exe.

FASTCALL

```
.globl @MyFunction1@8
.def    @MyFunction1@8; .scl    2;      .type   32;     .endef
@MyFunction1@8:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    %ecx, -4(%ebp)
        movl    %edx, -8(%ebp)
        movl    -4(%ebp), %eax
        leal    (%eax,%eax), %ecx
        movl    -8(%ebp), %edx
        movl    %edx, %eax
        addl    %eax, %eax
        addl    %edx, %eax
        leal    (%eax,%ecx), %eax
leave
ret
```

Notice first that the same name decoration is used as in cl.exe. The astute observer will already have realized that GCC uses the same trick as cl.exe, of moving the fastcall arguments from their registers (ecx and edx again) onto a negative offset on the stack. Again, optimizations are turned off. ecx is moved into the first position (-4) and edx is moved into the second position (-8). Like the CDECL example above, the value at -4 is doubled, and the value at -8 is tripled. Therefore, -4 (ecx) is x, and -8 (edx) is y. It would seem from this listing then that values are passed left-to-right, although we will need to take a look at the larger, MyFunction2 example:

```
.globl @MyFunction2@24
.def    @MyFunction2@24;        .scl    2;      .type   32;     .endef
@MyFunction2@24:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    %ecx, -4(%ebp)
        movl    %edx, -8(%ebp)
        movl    -4(%ebp), %eax
        imull   -8(%ebp), %eax
        movl    8(%ebp), %edx
        incl    %edx
        imull   %edx, %eax
        movl    12(%ebp), %edx
        addl    $2, %edx
        imull   %edx, %eax
        movl    16(%ebp), %edx
        addl    $3, %edx
        imull   %edx, %eax
        movl    20(%ebp), %edx
        addl    $4, %edx
        imull   %edx, %eax
leave
ret     $16
```

By following the fact that in MyFunction2, successive parameters are added to increasing constants, we can deduce the positions of each parameter. -4 is still x, and -8 is still y. +8 gets incremented by 1 (z), +12 gets increased by 2 (a). +16 gets increased by 3 (b), and +20 gets increased by 4 (c). Let’s list these values then:

```
z = [ebp + 8]
a = [ebp + 12]
b = [ebp + 16]
c = [ebp + 20]
```

c is the furthest down, and therefore was the first pushed. z is the highest to the top, and was therefore the last pushed. Arguments are therefore pushed in right-to-left order, just like cl.exe.

STDCALL

Let’s compare then the implementation of MyFunction1 in GCC:

```
.globl _MyFunction1@8
.def    _MyFunction1@8; .scl    2;      .type   32;     .endef
```

```
_MyFunction1@8:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        leal    (%eax,%eax), %ecx
        movl    12(%ebp), %edx
        movl    %edx, %eax

        addl    %eax, %eax

        addl    %edx, %eax
        leal    (%eax,%ecx), %eax
        popl    %ebp
        ret     $8
```

The name decoration is the same as in cl.exe, so STDCALL functions (and CDECL and FASTCALL for that matter) can be assembled with either compiler, and linked with either linker, it seems. The stack frame is set up, then the value at [ebp + 8] is doubled. After that, the value at [ebp + 12] is tripled. Therefore, +8 is x, and +12 is y. Again, these values are pushed in right-to-left order. This function also cleans it's own stack with the "ret 8" instruction.

Looking at a bigger example:

```
.globl _MyFunction2@24
.def    _MyFunction2@24;        .scl    2;      .type   32;     .endef
_MyFunction2@24:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    16(%ebp), %edx
        incl    %edx
        imull   %edx, %eax
        movl    20(%ebp), %edx
        addl    $2, %edx
        imull   %edx, %eax
        movl    24(%ebp), %edx
        addl    $3, %edx
        imull   %edx, %eax
        movl    28(%ebp), %edx
        addl    $4, %edx
        imull   %edx, %eax
        popl    %ebp
        ret     $24
```

We can see here that values at +8 and +12 from ebp are still x and y, respectively. The value at +16 is incremented by 1, the value at +20 is incremented by 2, etc all the way to the value at +28. We can therefore create the following table:

```
x = [ebp + 8]
y = [ebp + 12]
z = [ebp + 16]
a = [ebp + 20]
b = [ebp + 24]
c = [ebp + 28]
```

With c being pushed first, and x being pushed last. Therefore, these parameters are also pushed in right-to-left order. This function then also cleans 24 bytes off the stack with the “ret 24” instruction.

Calling Conventions Questions

Question 1

Identify the calling conventions of the following functions:

Question 1a

```
int Question1a(int a, int b)
{
return a + b;
}
```

Question 1b

```
:@Question1b@0
push ebp
mov ebp, esp
...
pop ebp
ret 12
```

Question 1c

```
push ebp
mov ebp, esp
add eax, edx
pop ebp
ret
```

Question 1d

```
push ebp
mov ebp, esp
mov eax, [ebp + 8]
pop ebp
ret 16
```

Calling Covnentions Answers

Answer 1

Question 1a

The function is written in C, and has no other specifiers, so it is CDECL by default.

Question 1b

The function includes the decorated name of an STDCALL function, and cleans up it's own stack. It is therefore an STDCALL function.

Question 1c

The function sets up a stack frame, so we know the compiler hasnt done anything "funny" to it. It accesses registers which arent initialized yet, in the edx and eax registers. It is therefore a FASTCALL function.

Question 1d

The function has a standard stack frame, and the *ret* instruction has a parameter to clean it's own stack. Also, it accesses a parameter from the stack. It is therefore an STDCALL function.

Variables

We've already seen some mechanisms to create local storage on the stack. This chapter will talk about some other variables, including **global variables**, **static variables**, variables labled "**const**," "**register**," and "**volatile**." It will also consider some general techniques concerning variables, including accessor and setter methods (to borrow from OO terminology). This section may also talk about setting memory breakpoints in a debugger to track memory I/O on a variable.

How to Spot a Variable

Variables come in 2 distinct flavors: those that are created on the stack (local variables), and those that are accessed via a hardcoded memory address (global variables). Any memory that is accessed via a hard-coded address is usually a global variable. Variables that are accessed as an offset from esp, or ebp are frequently local variables.

Hardcoded address
: Anything hardcoded is a value that is stored as-is in the binary, and is not changed at runtime. for instance, the value 0x2054 is hardcoded, whereas the current value of variable X is not hard-coded and may change at runtime.

Example of a hardcoded address:

```
mov eax, [0x77651010]
```

OR:

```
mov ecx, 0x77651010
mov eax, [ecx]
```

Example of a non-hardcoded (softcoded?) address:

```
mov ecx, [esp + 4]
add ecx, ebx
mov eax, [ecx]
```

Because in the last example, the value of ecx is calculated at run-time, whereas in the first 2 examples, the value is the same every time. RVAs are considered hard-coded addresses, even though the loader needs to "fix them up" to point to the correct locations.

.BSS and .DATA sections

Global Variables

"Static" Local Variables

Local variables labeled **static** maintain their value across function calls, and therefore cannot be created on the stack like other local variables are. How are static variables created? Let's take a simple example C function:

```
void MyFunction(int a)
{
        static int x = 0;
        printf("my number: ");
        printf("%d, %d\n", a, x);
}
```

Compiling to a listing file with **cl.exe** gives us the following code:

```
_BSS       SEGMENT
?x@?1??MyFunction@@9@9 DD 01H DUP (?)           ; `MyFunction'::`2'::x
_BSS    ENDS
_DATA   SEGMENT
$SG796  DB      'my number: ', 00H
$SG797  DB      '%d, %d', 0aH, 00H
_DATA   ENDS
PUBLIC  _MyFunction
EXTRN   _printf:NEAR
; Function compile flags: /Odt
_TEXT   SEGMENT
_a$ = 8                                 ; size = 4
_MyFunction PROC NEAR
; Line 4
        push    ebp
        mov     ebp, esp
; Line 6
        push    OFFSET FLAT:$SG796
        call    _printf
        add     esp, 4
; Line 7
        mov     eax, DWORD PTR ?x@?1??MyFunction@@9@9
        push    eax
        mov     ecx, DWORD PTR _a$[ebp]
        push    ecx
        push    OFFSET FLAT:$SG797
        call    _printf
        add     esp, 12                                 ; 0000000cH
; Line 8
        pop     ebp
        ret     0
```

```
_MyFunction ENDP
_TEXT   ENDS
```

Normally when assembly listings are posted in this , most of the code gibberish is discarded to aid readability, but in this instance, the "gibberish" contains the answer we are looking for. As can be clearly seen, this function creates a standard stack frame, and it doesn't create any local variables on the stack. In the interests of being complete, we will take baby-steps here, and work to the conclusion logically.

In the code for Line 7, there is a call to _printf with 3 arguments. Printf is a standard **libc** function, and it therefore can be assumed to be cdecl calling convention. Arguments are pushed, therefore, from right to left. Three arguments are pushed onto the stack before _printf is called:

- DWORD PTR ?x@?1??MyFunction@@9@9
- DWORD PTR _a$[ebp]
- OFFSET FLAT:$SG797

The second one, _a$[ebp] is partially defined in this assembly instruction:

```
_a$ = 8
```

And therefore _a$[ebp] is the variable located at offset +8 from ebp, or the first argument to the function. OFFSET FLAT:$SG797 likewise is declared in the assembly listing as such:

```
SG797      DB      '%d, %d', 0aH, 00H
```

If you have your ASCII table handy, you will notice that 0aH = 0x0A = '\n'. OFFSET FLAT:$SG797 then is the format string to our printf statement. Our last option then is the mysterious-looking "?x@?1??MyFunction@@9@9", which is defined in the following assembly code section:

```
_BSS       SEGMENT
?x@?1??MyFunction@@9@9 DD 01H DUP (?)
_BSS    ENDS
```

This shows that the Microsoft C compiler creates static variables in the .bss section. This might not be the same for all compilers, but the lesson is the same: local static variables are created and used in a very similar, if not the exact same, manner as global values. In fact, as far as the reverser is concerned, the two are usually interchangeable. Remember, the only real difference between static variables and global variables is the idea of "scope", which is only used by the compiler.

Constants

Variables qualified with the **const** keyword (in C) are frequently stored in the .data section of the executable. Constant values can be distinguished because they are initialized at the beginning of the program, and are never modified by the program itself. Constants are frequently stored in the .data section of the executable.

"Volatile" memory

In C and C++, variables can be declared "volatile," which tells the compiler that the memory location can be accessed from *external* or *concurrent* processes, and that the compiler should not perform any optimizations on the variable. For instance, if multiple threads were all accessing and modifying a single global value, it would be bad for the compiler to store that variable in a register sometimes, and flush it to memory infrequently. In general, Volatile memory must be flushed to memory after every calculation, to ensure that the most current version of the data is in memory when other processes come to look for it.

It is not always possible to determine from a disassembly listing whether a given variable is a volatile variable. However, if the variable is accessed frequently from memory, and its value is constantly updated in memory (especially if there are free registers available), that's a good hint that the variable might be volatile.

Simple Accessor Methods

An Accessor Method is a tool derived from OO theory and practice. In it's most simple form, an accessor method is a function that receives no parameters (or perhaps simply an offset), and returns the value of a variable. Accessor and Setter methods are ways to restrict access to certain variables. The only standard way to get the value of the variable is to use the Accessor.

Accessors can prevent some simple problems, such as out-of-bounds array indexing, and using unitialized data. Frequently, Accessors contain little or no error-checking.

Here is an example:

```
push ebp
mov ebp, esp
mov eax, [ecx + 8] ;THISCALL function, passes "this" pointer in ecx
mov esp, ebp
pop ebp
ret
```

Because they are so simple, Accessor methods are frequently heavily optimized (they generally don't need a stack frame), and are even occasionally *inlined* by the compiler.

Simple Setter (Manipulator) Methods

Setter methods are the antithesis of an Accessor method, and provide a unified way of altering the value of a given variable. Setter methods will often take as a parameter the value to be set to the variable, although some methods (Initializers) simply set the variable to a pre-defined value. Setter methods often do bounds checking, and error checking on the variable before it is set, and frequently either a) return no value, or b) return a simple boolean value to determine success.

Here is an example:

```
push ebp
mov ebp, esp
cmp [ebp + 8], 0
je error
mov eax, [ebp + 8]
mov [ecx + 0], eax
mov eax, 1
jmp end
:error
mov eax, 0
:end
mov esp, ebp
pop ebp
ret
```

Variables Questions

Question 1

Can you tell what the original C++ source code looks like, in general, for the following accessor method?

```
push ebp
mov ebp, esp
mov eax, [ecx + 8] ;THISCALL function, passes "this" pointer in ecx
mov esp, ebp
pop ebp
ret
```

You don't need to be specific, just provide the general outline.

Question 2

Can you tell what the original C++ source code looks like, in general, for the following setter method?

```
push ebp
mov ebp, esp
cmp [ebp + 8], 0
je error
mov eax, [ebp + 8]
mov [ecx + 0], eax
mov eax, 1
jmp end
:error
mov eax, 0
:end
mov esp, ebp
pop ebp
ret
```

There are conditional statements here, which may be difficult to translate at this point, so make a best guess as to what this function looks like.

Variables Answers

Answer 1

We don't know the name of the class, so we will use a generic name MyClass (or whatever you would like to call it). We will lay out a simple class definition, that contains a data value at offset +8. Offset +8 is the only data value accessed, so we don't know what the first 8 bytes of data looks like, but we will just assume (for our purposes) that our class looks like this:

```
class MyClass
{
  int value1;
  int value2;
  int value3; //offset +8
  ...
}
```

We will then create our function, which I will call "GetValue3()". We know that the data value being accessed is located at [ecx+8], (which we have defined above to be "value3"). Also, we know that the data is being read into a 4-byte register (eax), and is not truncated. We can assume, therefore, that value3 is a 4-byte data value. We can use the **this** pointer as the pointer value stored in ecx, and we can take the element that is at offset +8 from that pointer (value3):

```
MyClass::GetValue3()
{
  return this.value3;
}
```

The **this** pointer is not necessary here, but i use it anyway to illustrate the fact that the variable was accessed as an offset from the **this** pointer.

Note: Remember, we don't know what the first 8 bytes actually look like in our class, we only have a single accessor method, that only accesses a single data value at offset +8. The class could also have looked like this:

```
class MyClass /*Alternate Definition*/
{
   byte byte1;
   byte byte2;
   short short1;
   long value2;
   long value3;
 ...
}
```

Or, any other combinations of 8 bytes.

Answer 2

This code looks a little complicated, but don't panic! We will walk through it slowly. The first two lines of code set up the stack frame:

```
push ebp
mov ebp, esp
```

The next two lines of code compare the value of [ebp + 8] (which we know to be the first parameter) to zero. If [ebp+8] is zero, the function jumps to the label "error". We see that the label "error" sets eax to 0, and returns. We haven't seen it before, but this looks conspicuously like an **if** statement. "If the parameter is zero, return zero".

If, on the other hand, the parameter is not zero, we move the value into eax, and then move the value into [ecx + 0], which we know as the first data field in MyClass. We also see, from this code, that this first data field must be 4 bytes long (because we are using eax). After we move eax into [ecx + 0], we set eax to 1 and jump to the end of the function.

If we use the same MyClass defintion as in question 1, above, we can get the following code for our function, "SetValue1(int val)":

```
int MyClass::SetValue1(int val)
{
  if(val == 0) return 0;
  this.value1 = val;
  return 1;
}
```

Notice that since we are returning a 0 on failure, and a 1 on success, the function looks like it has a BOOL return value. However, because the return value is 4-bytes wide (eax is used), we know it can't be a BOOL (which is usually defined to be 1 byte long).

Don't get discouraged by the **if** statement, we talk about that in a later chapter.

Data Structures

Few programs can work by using simple memory storage, most need to utilize complex data objects, including **pointers**, **arrays**, **structures**, and other complicated types. This chapter will talk about how compilers implement complex data objects, and how the reverser can identify these objects.

Arrays

Arrays are simply a storage scheme for multiple data objects of the same type. Data objects are stored sequentially, often as an offset from a pointer to the beginning of array. Consider the following C code:

```
x = array[25];
```

Which is identical to the following asm code:

```
mov ebx, $array
mov eax, [ebx + 25]
mov $x, eax
```

Now, consider the following example:

```
int MyFunction1()
{
int array[20];
   ...
```

This (roughly) translates into the following asm code:

```
:_MyFunction1
push ebp
mov ebp, esp
sub esp, 80 ;the whole array is created on the stack!!!
lea $array, [esp + 0] ;a pointer to the array is saved in the array variable
...
```

The entire array is created on the stack, and the pointer to the bottom of the array is stored in the variable “array”. An optimizing compiler could ignore the last instruction, and simply refer to the array via a +0 offset from esp (in this example), but we will do things verbosely.

Likewise, consider the following example:

```
void MyFunction2()
{
char buffer[4];
   ...
```

This will translate into the following asm code:

```
:_MyFunction2
push ebp
mov ebp, esp
sub esp, 4
lea $buffer, [esp + 0]
...
```

Which, looks harmless enough. But, what if a program inadvertantly accesses buffer? what about buffer? what about buffer? This is the makings of a buffer overflow vulnerability, and (might) will be discussed in a later section. However, this section won't talk about security issues, and instead will focus only on data structures.

To Spot an Array on the Stack: To spot an array on the stack, look for large amounts of local storage allocated on the stack ("sub esp, 1000", for example), and look for large portions of that data being accessed by an offset from a different register from esp. For instance:

```
:_MyFunction3
push ebp
mov ebp, esp
sub esp, 256
lea ebx, [esp + 0x00]
mov [ebx + 0], 0x00
```

Is a good sign of an array being created on the stack. Granted, an optimizing compiler might just want to offset from esp instead, so you will need to be careful.

To Spot an Array in Memory: Arrays in memory, such as global arrays, or arrays which have initial data (remember, initialized data is created in the .data section in memory) will be accessed as offsets from a hardcoded address in memory:

```
:_MyFunction4
push ebp
mov ebp, esp
mov esi, 0x77651004
mov ebx, 0x00000000
mov [esi + ebx], 0x00
```

It needs to be kept in mind that structures and classes might be accessed in a similar manner, so the reverser needs to remember that all the data objects in an array are of the same type, that they

are sequential, and they will often be handled in a loop of some sort. Also, (and this might be the most important part), each elements in an array may be accessed by a *variable offset from the base*.

Structures

All C programmers are going to be familiar with the following syntax:

```
struct MyStruct
{
int FirstVar;
double SecondVar;
unsigned short int ThirdVar;
}
```

It's called a **structure** (Pascal programmers may know a similar concept as a "record").

Structures may be very big or very small, and they may contain all sorts of different data. Structures may look very similar to arrays in memory, but a few key points need to be remembered: Structures do not need to contain data fields of all the same type, structure fields are often 4-byte aligned (not sequential), and each element in a structure has it's own offset. It therefore makes no sense to reference a structure element by a variable offset from the base.

Take a look at the following structure definition:

```
struct MyStruct2
{
long value1;
short value2;
long value3;
}
```

Assuming the pointer to the base of this structure is loaded into ebx, we can access these members in one of two schemes:

```
;data is 32-bit aligned
[ebx + 0] ;value1
[ebx + 4] ;value2
[ebx + 8] ;value3
```

OR:

```
;data is "packed"
[ebx + 0] ;value1
[ebx + 4] ;value2
[ebx + 6] ;value3
```

The first arrangement is the most common, but it clearly leaves open an entire memory word (2 bytes) at offset +6, which are not used at all. Compilers occasionally allow the programmer to manually specify the offset of each data member, but this isn't always the case. The second example also has the benefit that the reverser can easily identify that each data member in the structure is a different size.

Consider now the following function:

```
:_MyFunction
push ebp
mov ebp, esp
lea ecx, SS:[ebp + 8]
mov [ecx + 0], 0x0000000A
mov [ecx + 4], ecx
mov [ecx + 8], 0x0000000A
mov esp, ebp
pop ebp
```

The function clearly takes a pointer to a data structure as it's first argument. Also, each data member is the same size (4 bytes), so how can we tell if this is an array or a structure? To answer that question, we need to remember 1 important distinction between structures and arrays: The elements in an array are all of the same type, the elements in a structure do not need to be the same type. Given that rule, it is clear that one of the elements in this structure is a pointer (it points to the base of the structure itself!) and the other two fields are loaded with the hex value 0x0A (10 in decimal), which is certainly not a valid pointer on any system I have ever used. We can then partially recreate the structure and the function code below:

```
struct MyStruct3
{
long value1;
void *value2;
long value3;
}
void MyFunction2(struct MyStruct3 *ptr)
{
ptr->value1 = 10;
ptr->value2 = ptr;
ptr->value3 = 10;
}
```

As a quick aside note, notice that this function doesn't load anything into eax, and therefore it doesn't return a value.

Advanced Structures

Lets say we have the following situation in a function:

```
:MyFunction1
push ebp
```

```
mov ebp, esp

mov esi, [ebp + 8]

lea ecx, SS:[esi + 8]
...
```

what is happening here? First, esi is loaded with the value of the function's first parameter (ebp + 8). Then, ecx is loaded with a pointer to the offset +8 from esi. It looks like we have 2 pointers accessing the same data structure!

The function in question could easily be one of the following 2 prototypes:

```
struct MyStruct1
{
DWORD value1;
DWORD value2;
struct MySubStruct1
  {
     ...
```

OR:

```
struct MyStruct2
{
DWORD value1;
DWORD value2;
DWORD array[LENGTH];
   ...
```

one pointer offset from another pointer in a structure often means a complex data structure. There are far too many combinations of structures and arrays, however, so this will not spend too much time on this subject.

Objects and Classes

Object-Oriented (OO) programming provides for us a new unit of program structure to contend with: the **Object**. This chapter will look at disassembled classes, primarily from C++, but also from any other OO compiled languages. This chapter will not deal directly with COM, but it will work to set a lot of the groundwork for future discussions in reversing COM components (Windows users only).

Classes

Classes Vs. Structs

Branches

Computer science professors tell their students to avoid jumps and **goto** instructions, to avoid the proverbial "spaghetti code." Unfortunately, assembly only has jump instructions to control program flow. This chapter will explore the subject that many people avoid like the plague, and will attempt to show how the spaghetti of assembly can be translated into the more familiar control structures of high-level language. Specifically, this chapter will focus on **If-Then-Else** and **Switch** branching instructions.

If-Then

Let's take a look at a generic **if** statement:

```
if(x)
{
//conditional code
}
//code to be done after the conditional
```

What does this code do? In english, the code checks x, and *doesn't jump* if x is true. Conversely, the if statement does jump if x is false. In pseudo-code then, the previous if statement does the following:

```
if not x goto end
//conditional code
end:
//code to be done after the conditional
```

Now with that format in mind, let's take a look at some actual C code:

```
if(x == 0)
{
x = 1;
}
x++;
```

When we translate that to assembly, we need to *reverse* the conditional jump from a **je** to a **jne** because—like we said above—we only jump if the condition is false.

```
mov eax, $x
cmp eax, 0x00000000
jne end
mov eax, 1
end:
inc eax
mov $x, eax
```

When you see a comparison, followed by a je or a jne, **reverse the condition of the jump to recreate the high-level code.** For jump-if-greater (jg), jump-if-greater-or-equal (jge), jump-if-less-than (jl), or similar instructions, it is a bit different than simply reversing the condition of the jump. For example, this assembler code:

```
mov eax, $x                    //move x into eax
cmp eax, $y                    //compare eax with y
jg end                         //jump if greater than
inc eax
move $x, eax                   //increment x
end:
...
```

Is produced by these c statements:

```
if(x <= y)
{
x++;
}
```

As you can see, x is incremented only if it is **less than or equal to** y. Thus, if it is greater than y, it will not be incremented as in the assembler code. Similarly, the c code

```
if(x < y)
{
arg1++;
}
```

Produces this assembler code:

```
mov eax, $x                         //move x into eax
cmp eax, $y                         //compare eax with y
jge end                             //jump if greater than or equal to
inc eax
move $x, eax                        //increment x
end:
...
```

X is incremented in the c code only if it is **less than** y, so the assembler code now jumps if it's greater than or equal to y. This kind of thing takes practice, so we will try to include lots of examples in this section.

If-Then-Else

Let us now look at a more complicated case: the **If-Then-Else** instruction. Here is a generic example:

```
if(x)
{
//do this is x is true
}
else
{
//do this if x is false
}
//do this after the if statement is over
```

Now, what happens here? Like before, the if statement only jumps to the else clause when x is false. However, we must also install an *unconditional* jump at the end of the "then" clause, so we dont perform the else clause directly afterwards.

Here is the above example in pseudocode:

```
if not x goto else
//do this if x is true
goto end
else:
//do this if x is false
end:
//do this after the if statement is over
```

Now, here is an example of a real C If-Then-Else:

```
if(x == 10)
{
x = 0;
}
else
{
```

```
x++;
}
```

Which gets translated into the following assembly code:

```
mov eax, $x
cmp eax, 0x0A ;0x0A = 10
jne else
mov eax, 0
jmp end
else:
inc eax
end:
mov $x, eax
```

As you can see, the addition of a single unconditional jump can add an entire extra option to our conditional.

Switch-Case

Switch-Case structures can be very complicated when viewed in assembly language, so we will examine a few examples. First, keep in mind that in C, there are several keywords that are commonly used in a switch statement. Here is a recap:

Switch

This keyword tests the argument, and starts the switch structure

Case

This creates a label that execution will switch to, depending on the value of the argument.

Break

This statement jumps to the end of the switch block

Default

This is the label that execution jumps to if and only if it doesn't match up to any other conditions

Lets say we have a general switch statement, but with an extra label at the end, as such:

```
switch (x)
{
//body of switch statement
}
end_of_switch:
```

Now, every **break** statement will be immediately replaced with the statement

```
jmp end_of_switch
```

But what do the rest of the statements get changed to? The case statements can each resolve to any number of arbitrary integer values. How do we test for that? The answer is that we use a “Switch Table”. Here is a simple, C example:

```
int main(int argc, char **argv)
{ //line 10
switch(argc)
        {
case 1:
MyFunction(1);
break;
case 2:
MyFunction(2);
break;
case 3:
MyFunction(3);
break;
case 4:
MyFunction(4);
break;
default:
MyFunction(5);
        }
return 0;
}
```

And when we compile this with **cl.exe**, we can generate the following listing file:

```
tv64 = -4          ; size = 4
_argc$ = 8              ; size = 4
_argv$ = 12             ; size = 4
_main   PROC NEAR
; Line 10
        push    ebp
        mov     ebp, esp
        push    ecx
; Line 11
        mov     eax, DWORD PTR _argc$[ebp]
        mov     DWORD PTR tv64[ebp], eax
        mov     ecx, DWORD PTR tv64[ebp]
        sub     ecx, 1
        mov     DWORD PTR tv64[ebp], ecx
        cmp     DWORD PTR tv64[ebp], 3
        ja      SHORT $L810
        mov     edx, DWORD PTR tv64[ebp]
        jmp     DWORD PTR $L818[edx*4]
$L806:
; Line 14
        push    1
        call    _MyFunction
        add     esp, 4
; Line 15
jmp     SHORT $L803
$L807:
```

```
; Line 17

        push    2

        call    _MyFunction
        add     esp, 4
; Line 18
jmp     SHORT $L803
$L808:
; Line 19
        push    3
        call    _MyFunction
        add     esp, 4
; Line 20
jmp     SHORT $L803
$L809:
; Line 22
        push    4
        call    _MyFunction
        add     esp, 4
; Line 23
jmp     SHORT $L803
$L810:
; Line 25
        push    5
        call    _MyFunction
        add     esp, 4
$L803:
; Line 27
xor     eax, eax
; Line 28
        mov     esp, ebp
        pop     ebp
        ret     0
$L818:
        DD      $L806
        DD      $L807
        DD      $L808
        DD      $L809
_main   ENDP
```

Lets work our way through this. First, we see that line 10 sets up our standard stack frame, and it also saves ecx. Why does it save ecx? Scanning through the function, we never see a corresponding “pop ecx” instruction, so it seems that the value is never restored at all. In fact, the compiler isn’t saving ecx at all, but is instead simply reserving space on the stack: It’s creating a local variable. The original C code didnt have any local variables, however, so perhaps the compiler just needed some extra scratch space to store intermediate values. Why doesnt the compiler execute the more familar “sub esp, 4” command to create the local variable? **push ecx** is just a faster instruction, that does the same thing. This “scratch space” is being referenced by a *negative offset* from ebp. **tv64** was defined in the beginning of the listing as having the value -4, so every call to “tv64[ebp]” is a call to this scratch space.

There are a few things that we need to notice about the function in general:

- Label $L803 is the end_of_switch label. Therefore, every “jmp SHORT $L803” statement is a **break**. This is verifiable by comparing with the C code line-by-line.

- Label $L818 contains a list of hard-coded memory addresses, which here are labels in the code section! Remember, labels resolve to the memory address of the instruction. This must be an important part of our puzzle.

To solve this puzzle, we will take an indepth look at line 11:

```
mov        eax, DWORD PTR _argc$[ebp]
mov     DWORD PTR tv64[ebp], eax
mov     ecx, DWORD PTR tv64[ebp]
sub     ecx, 1
mov     DWORD PTR tv64[ebp], ecx
cmp     DWORD PTR tv64[ebp], 3
ja      SHORT $L810
mov     edx, DWORD PTR tv64[ebp]
jmp     DWORD PTR $L818[edx*4]
```

The Setup

```
mov        eax, DWORD PTR _argc$[ebp]
mov     DWORD PTR tv64[ebp], eax
mov     ecx, DWORD PTR tv64[ebp]
sub     ecx, 1
mov     DWORD PTR tv64[ebp], ecx
```

The value of argc is moved into eax. The value of eax is then immediately moved to the scratch space. The value of the scratch space is then moved into ecx. Sounds like an awfully convoluted way to get the same value into so many different locations, but remember: I turned off the optimizations. The value of ecx is then decremented by 1. Why didn't the compiler use a **dec** instruction instead? perhaps the statement is a general statement, that in this case just happens to have an argument of 1. We dont know why exactly, all we know is this:

- eax = "scratch pad"
- ecx = eax - 1

Finally, the last line moves the new, decremented value of ecx *back into the scratch pad*. Very inefficient.

The Compare and Jumps

```
cmp        DWORD PTR tv64[ebp], 3
ja      SHORT $L810
```

The value of the scratch pad is compared with the value 3, and if the *unsigned* value is above 3 (4 or more), execution jumps to label $L810. How do I know the value is unsigned? I know because **ja** is an unsigned conditional jump. Lets look back at the original C code switch:

```
switch(argc)
        {
case 1:
MyFunction(1);

break;

case 2:
MyFunction(2);
break;
case 3:
MyFunction(3);
break;
case 4:
MyFunction(4);
break;
default:
MyFunction(5);
        }
```

Remember, the scratch pad contains the value (argc -1), which means that this condition is only triggered when argc > 4. What happens when argc is greater than 4? The function goes to the default condition. Now, let's look at the next two lines:

```
mov        edx, DWORD PTR tv64[ebp]
jmp     DWORD PTR $L818[edx*4]
```

edx gets the value of the scratch pad (argc - 1), and then there is a very weird jump that takes place: execution jumps to a location pointed to by the value (edx * 4 + $L818). What is $L818? We will examine that right now.

The Switch Table

```
$L818:
        DD      $L806
        DD      $L807
        DD      $L808
        DD      $L809
```

$L818 is a pointer, in the code section, to a list of other code section pointers. These pointers are all 32bit values (DD is a DWORD). Let's look back at our jump statement:

```
jmp        DWORD PTR $L818[edx*4]
```

In this jump, $L818 *isnt the offset, it's the base*, edx*4 is the offset. As we said earlier, edx contains the value of (argc - 1). If argc == 1, we jump to [$L818 + 0] which is $L806. If argc == 2, we jump to [$L818 + 4], which is $L807. Get the picture? A quick look at labels $L806, $L807, $L808, and $L809 shows us exactly what we expect to see: the bodies of the **case**

statements from the original C code, above. Each one of the case statements calls the function “MyFunction”, then breaks, and then jumps to the end of the switch block.

Ternary Operator ?:

Again, the best way to learn is by doing. Therefore we will go through a mini example to explain the ternary operator. Consider the following C code program:

```
int main(int argc, char **argv)
{
return (argc > 1)?(5):(0);
}
```

cl.exe produces the following assembly listing file:

```
_argc$ = 8                                              ; size = 4
_argv$ = 12                                                  ; size = 4
_main   PROC NEAR
; File c:\documents and settings\andrew\desktop\test2.c
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        xor     eax, eax

       cmp      DWORD PTR _argc$[ebp], 1
       setle    al
       dec      eax
       and      eax, 5
; Line 4
        pop     ebp
        ret     0
_main   ENDP
```

Line 2 sets up a stack frame, and line 4 is a standard entry sequence. There are no local variables. It is clear that Line 3 is where we want to look.

The instruction “xor eax, eax” simply sets eax to 0. For more information on that line, see the chapter on unintuitive instructions. The **cmp** instruction tests the condition of the ternary operator. The **setle** function is one of a set of x86 functions that works like a conditional move: al gets the value 1 if argc <= 1. Isnt that the exact opposite of what we wanted? In this case, it is. Lets look what happens when argc = 0: **al** gets the value 1. **al** is decremented (al = 0), and then eax is logically anded with 5. 5 & 0 = 0. When argc == 2 (greater then 1), the setle instruction doesnt do anything, and eax still is zero. eax is then decremented, which means that eax == -1. What is -1?

In x86 processors, negative numbers are stored in **twos-complement** format. for instance, lets look at the following C code:

```
BYTE x;
x = -1;
```

At the end of this C code, **x** will have the value 11111111: all ones!

When argc is greater then 1, setle sets al to zero. decrementing this value sets every bit in eax to a logical 1. Now, when we perform the logical **and** function we get:

```
...11111111
&...00000101     ;101 is 5 in binary
```

```
...00000101
```

eax gets the value 5. In this case, it's a roundabout method of doing it, but as a reverser, this is the stuff you need to worry about.

For reference, here is the GCC assembly output of the same ternary operator from above:

```
_main:
pushl   %ebp
movl    %esp, %ebp
subl    $8, %esp
xorl    %eax, %eax
andl    $-16, %esp
call    __alloca
call    ___main
xorl    %edx, %edx
cmpl    $2, 8(%ebp)
setge   %dl
leal    (%edx,%edx,4), %eax
leave
ret
```

Notice that GCC produces slightly different code then cl.exe produces. However, the stack frame is set up the same way. Notice also that GCC doesnt give us line numbers, or other hints in the code. The ternary operator line occurs after the instruction "call __main". Let's highlight that section here:

```
xorl       %edx, %edx
cmpl    $2, 8(%ebp)
setge   %dl
leal    (%edx,%edx,4), %eax
```

Again, **xor** is used to set edx to 0 quickly. Argc is tested against 2 (instead of 1), and dl is set if argc is *greater then or equal*. If dl gets set to 1, the **leal** instruction directly thereafter will move the value of 5 into eax.

Branches Questions

Consider the following function:

```
push ebp
mov ebp, esp
mov eax, 0
mov ecx, [ebp + 8]
cmp ecx, 0
jne _Label_1
inc eax
jne _Label_2
:_Label_1
dec eac
: _Label_2
mov ecx, [ebp + 12]
cmp ecx, 0
jne _Label_3
inc eax
: _Label_3
mov esp, ebp
pop ebp
ret
```

Question 1

What parameters does this function take? What calling convention does it use? What kind of value does it return? Write the entire C prototype of this function. Assume all values are unsigned values.

Question 2

How many separate branch structures are in this function? What types are they? Can you give more descriptive names to _Label_1, _Label_2, and _Label_3, based on the structures of these branches?

Question 3

Write the equivalent C code for this function. Assume all parameters and return values are unsigned values.

Branches Answers

Answer 1

This function accesses parameters on the stack at [ebp + 8] and [ebp + 12]. Both of these values are loaded into ecx, and we can therefore assume they are 4-byte values. This function doesnt clean it’s own stack, and the values aren’t passed in registers, so we know the funcion is CDECL.

The return value in eax is a 4-byte value, and we are told to assume that all the values are unsigned. Putting all this together, we can construct the function prototype:

```
unsigned int CDECL MyFunction(unsigned int param1, unsigned int param2);
```

Answer 2

How many separate branch structures are there in this function? Stripping away the entry and exit sequences, here is the code we have left:

```
mov ecx, [ebp + 8]
cmp ecx, 0
jne _Label_1
inc eax
jne _Label_2
:_Label_1
dec eac
: _Label_2
mov ecx, [ebp + 12]
cmp ecx, 0
jne _Label_3
inc eax
: _Label_3
```

Looking through, we see 2 **cmp** statements. The first cmd statement compares ecx to zero. If ecx is not zero, we go to _Label_1, decrement eax, and then fall-through to _Label_2. If ecx is zero, we increment eax, and go to directly to _Label_2. Writing out some pseudocode, we have the following result for the first section:

```
if(ecx doesnt equal 0) goto _Label_1
eax++;
goto _Label_2
:_Label_1
eax--;
:_Label_2
```

Since _Label_2 occurs at the end of this structure, we can rename it to something more descriptive, like “End_of_Branch_1”, or “Branch_1_End”. The first comparison tests ecx against 0, and then jumps on not-equal. We can reverse the conditional, and say that _Label_1 is an **else** block:

```
if(ecx == 0) ;ecx is param1 here
{
eax++;
}
else
{
eax--;
}
```

So we can rename _Label_1 to something else descriptive, such as “Else_1”. The rest of the code block, after Branch_1_End (_Label_2) is as follows:

```
mov ecx, [ebp + 12]
cmp ecx, 0
jne _Label_3
inc eax
: _Label_3
```

We can see immediately that _Label_3 is the end of this branch structure, so we can immediately call it “Branch_2_End”, or something else. Here, we are again comparing ecx to 0, and if it is not equal, we jump to the end of the block. If it is equal to zero, however, we increment eax, and then fall out the bottom of the branch. We can see that there is no **else** block in this branch structure, so we don’t need to invert the condition. We can write an **if** statement directly:

```
if(ecx == 0) ;ecx is param2 here
{
eax++;
}
```

Answer 3

Starting with the C function prototype from answer 1, and the conditional blocks in answer 2, we can put together a pseudo-code function, without variable declarations, or a return value:

```
unsigned int CDECL MyFunction(unsigned int param1, unsigned int param2)
{
if(param1 == 0)
   {
eax++;
   }
else
   {
eax--;
   }
if(param2 == 0)
   {
eax++;
   }
}
```

Now, we just need to create a variable to store the value from eax, which we will call “a”, and we will declare as a **register** type:

```
unsigned int CDECL MyFunction(unsigned int param1, unsigned int param2)
{
register unsigned int a = 0;
if(param1 == 0)
   {
a++;
   }
```

```
else
   {
a--;
   }

if(param2 == 0)

   {
a++;
   }
return a;
}
```

Granted, this function isnt a particularly useful function, but at least we know what it does.

Loops

To complete repetitive tasks, programmers often implement **loops**. There are many sorts of loops, but they can all be boiled down to a few similar formats in assembly code. This chapter will discuss loops, how to identify them, and how to “decompile” them back into high-level representations.

Do-While Loops

It seems counterintuitive that this section will consider **Do-While** loops first, considering that they might be the least used of all the variations in practice. However, there is method to our madness, so read on.

Consider the following generic Do-While loop:

```
do
{
//loop body
} while(x);
```

What does this loop do? the loop body simply executes, the condition is tested at the end of the loop, and the loop jumps back to the beginning of the loop if the condition is satisfied. Unlike **if** statements, Do-While conditions are not reversed.

Let us now take a look at the following C code:

```
do
{
x++;
} while(x != 10);
```

Which can be translated into assembly language as such:

```
mov eax, $x
:beginning
inc eax
cmp eax, 0x0A ;0x0A = 10
jne beginning
mov $x, eax
```

While Loops

While loops look almost as simple as a **Do-While** loop, but in reality they aren’t as simple at all. Let’s examine a generic while-loop:

```
while(x)
{
//loop body
}
```

What does this loop do? First, the loop checks to make sure that x is true. If x is not true, the loop is skipped. The loop body is then executed, followed by another check: is x still true? If x is still true, execution jumps back to the top of the loop, and execution continues. Keep in mind that there needs to be a jump at the bottom of the loop (to get back up to the top), but it makes no sense to jump back to the top, retest the conditional, and then jump *back to the bottom of the loop* if the conditional is found to be false. The while-loop then, performs the following steps:

1. check the condition. if it is false, go to the end
2. perform the loop body
3. check the condition, if it is true, jump to 2.
4. if the condition is not true, fall-through the end of the loop.

Here is a while-loop in C code:

```
while(x <= 10)
{
x++;
}
```

And here then is that same loop translated into assembly:

```
mov eax, $x
cmp x, 0x0A
jg end
beginning:
inc eax
cmp eax, 0x0A
jle beginning
end:
```

If we were to translate that assembly code **back into C**, we would get the following code:

```
if(x <= 10) //remember: in If statements, we reverse the condition from the asm
{
do
  {
x++;
} while(x <= 10)
}
```

See why we covered the Do-While loop first? because the While-loop becomes a Do-While when it gets assembled.

For Loops

What is a For-Loop? in essence, it's a While-Loop with an initial state, a condition, and an iterative instruction. For instance, the following generic For-Loop:

```
for(A; B; C)
{
//loop body
}
```

Gets translated into the following while-loop:

```
A;
while(B)
{
//loop body
C;
}
```

Which in turn gets translated into the following Do-While Loop:

```
A;
if(B)
{
do
   {
//loop-body
C;
} while(B);
}
```

Other Loop Types

C only has Do-While, While, and For Loops, but some other languages may very well implement their own types. Also, a good C-Programmer could easily "home brew" a new type of loop using a series of good macros, so they bear some consideration:

Do-Until Loop

A common Do-Until Loop will take the following form:

```
do
{
//loop body
```

```
} until(x);
```

Which essentially becomes the following Do-While loop:

```
do
{
//loop body
} while(!x);
```

Until Loop

Like the Do-Until loop, the standard Until-Loop looks like the following:

```
until(x)
{
//loop body
}
```

Which (likewise) gets translated to the following While-Loop:

```
while(!x)
{
//loop body
}
```

Do-Forever Loop

A Do-Forever loop is simply an unqualified loop with a condition that is always true. For instance, the following pseudo-code:

```
doforever
{
//loop body
}
```

will become the following while-loop:

```
while(1)
{
//loop body
}
```

Which can actually be reduced to a simple unconditional jump statement:

```
beginning:
;loop body
jmp beginning
```

Loops Questions

Let’s say that we have the following assembly function:

```
push ebp
mov ebp, esp
mov esi, [ebp + 8]
mov ebx, 0
mov eax, 0
mov ecx, 0
_Label_1:
mov ecx, [esi + ebx * 4]
add eax, ecx
add ebx, 4
inc ebx
cmp ebx, 100
je _Label_1
mov esp, ebp
pop ebp
ret 4
```

Question 1

What does this function do? What kinds of parameters does it take, and what kind of results (if any) does it return?

Question 2

What is this functions C prototype? Make sure to include parameters, return values, and calling convention.

Question 3

Decompile this code into equivalent C source code.

Loops Answers

Answer 1

This function loops through an array of 4 byte integer values, pointed to by esi, and adds each entry. It returns the sum in eax. The only parameter (located in [ebp + 8]) is a pointer to an array of integer values. The comparison between ebx and 100 indicates that the input array has 100 entries in it. The pointer offset [esi + ebx * 4] shows that each entry in the array is 4 bytes wide.

Answer 2

Notice how the **ret** function cleans it's parameter off the stack? That means that this function is an STDCALL function. We know that the function takes, as it's only parameter, a pointer to an array of integers. We do not know, however, whether the integers are signed or unsigned, because the **je** command is used for both types of values. We can assume one or the other, and for simplicity, we can assume unsigned values (unsigned and signed values, in this function, will actually work the same way). We also know that the return value is a 4-byte integer value, of the same type as is found in the parameter array. Since the function doesnt have a name, we can just call it "MyFunction", and we can call the parameter "array" because it is an array. From this information, we can determine the following prototype in C:

```
unsigned int STDCALL MyFunction(unsigned int *array);
```

Answer 3

Starting with the function prototype above, and the description of what this function does, we can start to write the C code for this function. We know that this function initializes eax, ebx, and ecx before the loop. However, we can see that ecx is being used as simply an intermediate storage location, receiving successive values from the array, and then being added to eax.

We will create two unsigned integer values, a (for eax) and b (for ebx). We will define both a and b with the **register** qualifier, so that we can instruct the compiler not to create space for them on the stack. For each loop iteration, we are adding the value of the array, at location ebx*4 to the running sum, eax. Converting this to our a and b variables, and using C syntax, we see:

```
a = a + array[b];
```

The loop could be either a **for** loop, or a **while** loop. We see that the loop control variable, b, is initialized to 0 before the loop, and is incremented by 1 each loop iteration. The loop tests b against 100, *after it gets incremented*, so we know that b never equals 100 inside the loop body. Using these simple facts, we will write the loop in 3 different ways:

First, with a **while** loop.

```
unsigned int STDCALL MyFunction(unsigned int *array)
{
register unsigned int b = 0;
register unsigned int a = 0;
while(b != 100)
   {
a = a + array[b];
b++;
   }
```

```
return b;
}
```

Or, with a **for** loop:

unsigned int STDCALL MyFunction(unsigned int *array)

```
{
register unsigned int b;
register unsigned int a = 0;
for(b = 0; b != 100; b++)
   {
a = a + array[b];
   }
return b;
}
```

And finally, with a **do-while** loop:

unsigned int STDCALL MyFunction(unsigned int *array)

```
{
register unsigned int b = 0;
register unsigned int a = 0;
do
   {
a = a + array[b];
b++;
}while(b != 100);
return b;
}
```

Advanced Disassembly

This section is tasked with covering, at least in a broad sense, all topics that are necessary for a good reverser, but do not fit in the scope of the previous section on basic disassembly. Readers who have not read the section on Program Structure should make sure they have a firm footing in those fundamentals before reading chapters in this section.

Note on the Section

Some of these topics will eventually be covered in greater detail in later sections. Specifically, Code Obfuscation and Code Optimization are proposed to get their own section later.

Floating Point Numbers

This page will talk about how **floating point** numbers are used in assembly language constructs. This page will not talk about new constructs, it will not explain what the FPU instructions do, how floating point numbers are stored or manipulated, or the differences in floating-point data representations. However, this page will demonstrate briefly how floating-point numbers are used in code and data structures that we have already considered.

Calling Conventions

With the addition of the floating-point stack, there is an entirely new dimension for passing parameters, and returning values. We will examine our calling conventions here, and see how they are affected by the presence of floating-point numbers. These are the functions that we will be assembling, using both GCC, and cl.exe:

```
__cdecl double MyFunction1(double x, double y, float z)
{
return (x + 1.0) * (y + 2.0) * (z + 3.0);
}

__fastcall double MyFunction2(double x, double y, float z)
{
return (x + 1.0) * (y + 2.0) * (z + 3.0);
}

__stdcall double MyFunction3(double x, double y, float z)
{
return (x + 1.0) * (y + 2.0) * (z + 3.0);
}
```

Note: cl.exe doesnt use these directives, so to create these functions, 3 different files need to be created, compiled with the /Gd, /Gr, and /Gz options, respectively.

CDECL

Here is the cl.exe assembly listing for MyFunction1:

```
PUBLIC     _MyFunction1
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR
;       COMDAT __real@3ff0000000000000
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1
CONST   ENDS
;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r   ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
```

```
CONST   SEGMENT

__real@4008000000000000 DQ 04008000000000000r   ; 3

CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 8
_y$ = 16                                                ; size = 8
_z$ = 24                                                ; size = 4
_MyFunction1 PROC NEAR
; Line 2
        push    ebp
        mov     ebp, esp
; Line 3
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 4
        pop     ebp
        ret     0
_MyFunction1 ENDP
_TEXT   ENDS
```

Our first question is this: are the parameters passed on the stack, or on the floating-point register stack, or some place different entirely? Key to this question, and to this function is a knowledge of what **fld** and **fstp** do. fld (Floating-point Load) pushes a floating point value onto the FPU stack, while fstp (Floating-Point Store and Pop) moves a floating point value from ST0 to the specified location, and then pops the value from ST0 off the stack entirely. Remember that **double** values in cl.exe are treated as 8-byte storage locations (QWORD), while floats are only stored as 4-byte quantities (DWORD). It is also important to remember that floating point numbers are not stored in a human-readable form in memory, even if the reader has a solid knowledge of binary. Remember, these arent integers. Unfortunately, the exact format of floating point numbers is well beyond the scope of this chapter.

x is offset +8, y is offset +16, and z is offset +24 from ebp. Therefore, z is pushed first, x is pushed last, and the parameters are passed right-to-left on the *regular stack* not the floating point stack. To understand how a value is returned however, we need to understand what **fmulp** does. fmulp is the “Floating-Point Multiply and Pop” instruction. It performs the instructions:

```
ST1 := ST1 * ST0
FPU POP ST0
```

So the top 2 values are multipled together, and the result is stored on the top of the stack. Therefore, in our instruction above, “fmulp ST(1), ST(0)”, which is also the last instruction of the function, we can see that the last result is stored in ST0. Therefore, floating point parameters are passed on the regular stack, but floating point results are passed on the FPU stack.

One final note is that MyFunction2 cleans it's own stack, as referenced by the **ret 20** command at the end of the listing. Because none of the parameters were passed in registers, this function appears to be exactly what we would expect an STDCALL function would look like: parameters passed on the stack from right-to-left, and the function cleans it's own stack. We will see below that this is actually a correct presumption.

For comparison, here is the GCC listing:

```
LC1:
        .long   0
        .long   1073741824
.align 8
LC2:
        .long   0
        .long   1074266112
.globl _MyFunction1
.def    _MyFunction1;   .scl    2;      .type   32;     .endef
_MyFunction1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC1
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC2
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
leave
ret
.align 8
```

This is a very difficult listing, so we will step through it (albeit quickly). 16 bytes of extra space is allocated on the stack. Then, using a combination of fldl and fstpl instructions, the first 2 parameters are moved from offsets +8 and +16, to offsets -8 and -16 from ebp. Seems like a waste of time, but remember, optimizations are off. **fld1** loads the floating point value 1.0 onto the FPU stack. **faddp** then adds the top of the stack (1.0), to the value in ST1 ([ebp - 8], originally [ebp + 8]).

FASTCALL

Here is the cl.exe listing for MyFunction2:

```
PUBLIC     @MyFunction2@20
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR
```

```
;       COMDAT __real@3ff0000000000000
CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1

CONST   ENDS

;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r   ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
CONST   SEGMENT
__real@4008000000000000 DQ 04008000000000000r   ; 3
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                                 ; size = 8
_y$ = 16                                                ; size = 8
_z$ = 24                                                ; size = 4
@MyFunction2@20 PROC NEAR
; Line 7
        push    ebp
        mov     ebp, esp
; Line 8
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 9
        pop     ebp
        ret     20                                      ; 00000014H
@MyFunction2@20 ENDP
_TEXT   ENDS
```

We can see that this function is taking 20 bytes worth of parameters, because of the @20 decoration at the end of the function name. This makes sense, because the function is taking two **double** parameters (8 bytes each), and one **float** parameter (4 bytes each). This is a grand total of 20 bytes. We can notice at a first glance, without having to actually analyze or understand any of the code, that there is only one register being accessed here: **ebp**. This seems strange, considering that FASTCALL passes it's regular 32-bit arguments in registers. However, that is not the case here: all the floating-point parameters (even z, which is a 32-bit float) are passed on the stack. We know this, because by looking at the code, there is no other place where the parameters could be coming from.

Notice also that **fmulp** is the last instruction performed again, as it was in the CDECL example. We can infer then, without investigating too deeply, that the result is passed at the top of the floating-point stack.

Notice also that x (offset [ebp + 8]), y (offset [ebp + 16]) and z (offset [ebp + 24]) are pushed in reverse order: z is first, x is last. This means that floating point parameters are passed in right-to-left order, on the stack. This is exactly the same as CDECL code, although only because we are using floating-point values.

Here is the GCC assembly listing for MyFunction2:

```
.align 8
LC5:

        .long   0

        .long   1073741824
.align 8
LC6:
        .long   0
        .long   1074266112
.globl @MyFunction2@20
.def    @MyFunction2@20;        .scl    2;      .type   32;     .endef
@MyFunction2@20:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC5
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC6
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
leave
ret     $20
```

This is a tricky peice of code, but luckily we don't need to read it very close to find what we are looking for. First off, notice that no other registers are accessed besides **ebp**. Again, GCC passes all floating point values (even the 32-bit float, z) on the stack. Also, the floating point result value is passed on the top of the floating point stack.

We can see again that GCC is doing something strange at the beginning, taking the values on the stack from [ebp + 8] and [ebp + 16], and moving them to locations [ebp - 8] and [ebp - 16], respectively. Immediately after being moved, these values are loaded onto the floating point stack and arithmetic is performed. z isn't loaded till later, and isn't ever moved to [ebp - 24], despite the pattern.

LC5 and LC6 are constant values, that most likely represent floating point values (because the numbers themselves, 1073741824 and 1074266112 don't make any sense in the context of our example functions. Notice though that both LC5 and LC6 contain two **.long** data items, for a total of 8 bytes of storage? they are therefore most definately **double** values.

STDCALL

Here is the cl.exe listing for MyFunction3:

```
PUBLIC     _MyFunction3@20
PUBLIC  __real@3ff0000000000000
PUBLIC  __real@4000000000000000
PUBLIC  __real@4008000000000000
EXTRN   __fltused:NEAR

;       COMDAT __real@3ff0000000000000

CONST   SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r   ; 1
CONST   ENDS
;       COMDAT __real@4000000000000000
CONST   SEGMENT
__real@4000000000000000 DQ 04000000000000000r   ; 2
CONST   ENDS
;       COMDAT __real@4008000000000000
CONST   SEGMENT
__real@4008000000000000 DQ 04008000000000000r   ; 3
CONST   ENDS
_TEXT   SEGMENT
_x$ = 8                                         ; size = 8
_y$ = 16                                                ; size = 8
_z$ = 24                                                ; size = 4
_MyFunction3@20 PROC NEAR
; Line 12
        push    ebp
        mov     ebp, esp
; Line 13
        fld     QWORD PTR _x$[ebp]
        fadd    QWORD PTR __real@3ff0000000000000
        fld     QWORD PTR _y$[ebp]
        fadd    QWORD PTR __real@4000000000000000
        fmulp   ST(1), ST(0)
        fld     DWORD PTR _z$[ebp]
        fadd    QWORD PTR __real@4008000000000000
        fmulp   ST(1), ST(0)
; Line 14
        pop     ebp
        ret     20                                      ; 00000014H
_MyFunction3@20 ENDP
_TEXT   ENDS
END
```

x is the highest on the stack, and z is the lowest, therefore these parameters are passed from right-to-left. We can tell this because x has the smallest offset (offset [ebp + 8]), while z has the largest offset (offset [ebp + 24]). We see also from the final fmulp instruction that the return value is passed on the FPU stack. This function also cleans the stack itself, as noticed by the call '*ret 20*. It is cleaning exactly 20 bytes off the stack which is, incidentally, the total amount that we passed to begin with. We can also notice that the implementation of this function looks exactly like the FASTCALL version of this function. This is true because FASTCALL only passes DWORD-sized parameters in registers, and floating point numbers do not qualify. This means that our assumption above was correct.

Here is the GCC listing for MyFunction3:

```
.align 8
LC9:
        .long   0
        .long   1073741824
```

```
.align 8
LC10:
        .long   0
        .long   1074266112
.globl @MyFunction3@20
.def    @MyFunction3@20;        .scl    2;      .type   32;     .endef

@MyFunction3@20:

        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        fldl    8(%ebp)
        fstpl   -8(%ebp)
        fldl    16(%ebp)
        fstpl   -16(%ebp)
        fldl    -8(%ebp)
fld1
        faddp   %st, %st(1)
        fldl    -16(%ebp)
        fldl    LC9
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
        flds    24(%ebp)
        fldl    LC10
        faddp   %st, %st(1)
        fmulp   %st, %st(1)
leave
ret     $20
```

Here we can also see, after all the opening nonsense, that [ebp - 8] (originally [ebp + 8]) is value x, and that [ebp - 24] (originally [ebp - 24]) is value z. These parameters are therefore passed right-to-left. Also, we can deduce from the final fmulp instruction that the result is passed in ST0. Again, the STDCALL function cleans it's own stack, as we would expect.

Conclusions

Floating point values are passed as parameters on the stack, and are passed on the FPU stack as results. Floating point values do not get put into the general-purpose integer registers (eax, ebx, etc...), so FASTCALL functions that only have floating point parameters collapse into STDCALL functions instead. **double** values are 8-bytes wide, and therefore will take up 8-bytes on the stack. **float** values however, are only 4-bytes wide.

Float to Int Conversions

FPU Compares and Jumps

Code Optimization

An **optimizing compiler** is perhaps one of the most complicated, most powerful, and most interesting programs in existence. This chapter will talk about optimizations, although this

chapter will not include a table of common optimizations. Such a table will eventually be included in the chapter on Unintuitive Instructions

Stages of Optimizations

There are two times when a compiler can perform optimizations: first, in the intermediate representation, and second, during the code generation.

Intermediate Representation Optimizations

While in the intermediate representation, a compiler can perform various optimizations, often based on dataflow analysis techniques. For example, consider the following code fragment:

```
x = 5;
if(x != 5)
{
//loop body
}
```

An optimizing compiler might notice that at the point of “if (x != 5)”, the value of x is always the constant “5”. This allows substituting “5” for x resulting in “5 != 5”. Then the compiler notices that the resulting expression operates entirely on constants, so the value can be calculated now instead of at run time, resulting in optimizing the conditional to “if (false)”. Finally the compiler sees that this means the body of the if conditional will never be executed, so it can omit the entire body of the if conditional altogether.

Consider the reverse case:

```
x = 5;
if(x == 5)
{
//loop body
}
```

In this case, the optimizing compiler would notice that the If conditional will always be true, and it wont even bother writing code to test x.

Code Generation Optimizations

Once the compiler has sifted through all the logical inefficiencies in your code, the Code generator takes over. Often the code generator will replace certain slow machine instructions with faster machine instructions.

For instance, the instruction:

```
beginning:
...
loopnz beginning
```

operates *much* slower then the equivalent instruction set:

```
beginning:
...
dec ecx
jne beginning
```

So then why would a compiler ever use a loopxx instruction? The answer is that most optimizing compilers never use a loopxx instruction, and therefore as a reverser, you will probably never see one used in real code.

What about the instruction:

```
mov eax, 0
```

The mov instruction is relatively quick, but a faster part of the processor is the arithmetic unit. Therefore, it makes more sense to use the following instruction:

```
xor eax, eax
```

because xor operates in very few processor cycles (and saves a byte or two at the same time), and is therefore faster than a “mov eax, 0”.

The topic of code-generation-optimizations will be considered further in the chapter on Unintuitive Instructions.

Example 1: Optimized vs Non-Optimized Code

The best way to explain optimization is to show an example, so I have one prepared. The following example is adapted from an algorithm presented in Knuth(vol 1, chapt 1) used to find the greatest common denominator of 2 integers:

```
/*line 1*/
int EuclidsGCD(int m, int n) /*we want to find the GCD of m and n*/
{
int q, r; /*q is the quotient, r is the remainder*/
while(1)
        {
q = m / n; /*find q and r*/
r = m % n;
```

```
if(r == 0) /*if r is 0, return our n value*/
                {
return n;
                }
m = n; /*set m to the current n value*/
n = r; /*set n to our current remainder value*/
} /*repeat*/

}
```

this is a relatively simple example. Compiling with the Microsoft C compiler, we generate a listing file using no optimization:

```
PUBLIC     _EuclidsGCD
_TEXT   SEGMENT
_r$ = -8        ; size = 4
_q$ = -4        ; size = 4
_m$ = 8 ; size = 4
_n$ = 12        ; size = 4
_EuclidsGCD PROC NEAR
; Line 2
        push    ebp
        mov     ebp, esp
        sub     esp, 8
$L477:
; Line 4
        mov     eax, 1
        test    eax, eax
        je      SHORT $L473
; Line 6
mov     eax, DWORD PTR _m$[ebp]
cdq
        idiv    DWORD PTR _n$[ebp]
        mov     DWORD PTR _q$[ebp], eax
; Line 7
mov     eax, DWORD PTR _m$[ebp]
cdq
        idiv    DWORD PTR _n$[ebp]
        mov     DWORD PTR _r$[ebp], edx
; Line 8
        cmp     DWORD PTR _r$[ebp], 0
        jne     SHORT $L479
; Line 10
        mov     eax, DWORD PTR _n$[ebp]
        jmp     SHORT $L473
$L479:
; Line 12
        mov     ecx, DWORD PTR _n$[ebp]
        mov     DWORD PTR _m$[ebp], ecx
; Line 13
        mov     edx, DWORD PTR _r$[ebp]
        mov     DWORD PTR _n$[ebp], edx
; Line 14
jmp     SHORT $L477
$L473:
; Line 15
        mov     esp, ebp
        pop     ebp
        ret     0
_EuclidsGCD ENDP
_TEXT   ENDS
END
```

Notice how there is a very clear correspondence between the lines of C code, and the lines of the ASM code. the addition of the "; line x" directives is very helpful in that respect.

Next, we compile the same function using a series of optimizations to stress speed over size:

```
cl.exe /Tceuclids.c /Fa /Ogt2
```

and we produce the following listing:

```
PUBLIC     _EuclidsGCD
_TEXT   SEGMENT
_m$ = 8 ; size = 4
_n$ = 12       ; size = 4
_EuclidsGCD PROC NEAR
; Line 7
        mov     eax, DWORD PTR _m$[esp-4]
        push    esi
        mov     esi, DWORD PTR _n$[esp]
cdq
        idiv    esi
        mov     ecx, edx
; Line 8
        test    ecx, ecx
        je      SHORT $L563
$L547:
; Line 12
mov     eax, esi
cdq
idiv    ecx
; Line 13
        mov     esi, ecx
        mov     ecx, edx
        test    ecx, ecx
        jne     SHORT $L547
$L563:
; Line 10
        mov     eax, esi
        pop     esi
; Line 15
ret     0
_EuclidsGCD ENDP
_TEXT   ENDS
END
```

As you can see, the optimized version is significantly shorter then the non-optimized version. Some of the key differences include:

- The optimized version does not prepare a standard stack frame. This is important to note, because many times new reversers assume that functions always start and end with proper stack frames, and this is clearly not the case. EBP isnt being used, ESP isnt being altered (because the local variables are kept in registers, and not put on the stack), and no subfunctions are called. 5 instructions are cut by this.
- The "test EAX, EAX" series of instructions in the non-optimized output, under ";line 4" is all unnecessary. The while-loop is defined by "while(1)" and therefore the loop always continues. this extra code is safely cut out. Notice also that there

is no unconditional jump in the loop like would be expected: the “if(r == 0) return n;” instruction has become the new loop condition.

- The structure of the function is altered greatly: the division of m and n to produce q and r is performed in this function twice: once at the beginning of the function to initialize, and once at the end of the loop. Also, the value of r is tested twice, in the same places. The compiler is very liberal with how it assigns storage in the function, and readily discards values that are not needed.

Optimization Questions

Question 1

The operations of line 4 from the non-optimized assembly code from Example 1 can be easily condensed to a single instruction. Can you “optimize” that section?

Question 2

In the optimized assembly listing, which register holds r, and which register holds q?

Question 3

Can you “decompile” the *optimized* assembly listing above into an “optimized” version of our original C code that mimics the same structure and algorithm?

Question 4

Why does the dec/jne combo operate faster than the equivalent loopnz?

Optimization Answers

Answer 1

The code in this line is the code generated for the “while(1)” C code, to be exact, it represents the loop break condition. Because this is an infinite loop, we can assume that these lines are unnecessary.

“mov eax, 1” initializes eax.

the test immediately afterwards tests the value of eax to ensure that it is nonzero. because eax will always be nonzero (eax = 1) at this point, the conditional jump can be removed along whith the “mov” and the “test”.

The assembly is actully checking whether 1 equals 1. Another fact is, that the C code for an infinite **FOR** loop:

```
for( ; ; )
{
   ...
}
```

would not create such a meaningless assembly code to begin with, and is logically the same as “while(1)”.

Answer 2

At the beginning of the function, **eax** contains m, and **esi** contains n. When the instruction “idiv esi” is executed, eax contains the quotient (q), and edx contains the remainder ®. The instruction “mov ecx, edx” moves r into **ecx**, while q is not used for the rest of the loop, and is therefore discarded.

Answer 3

Altering the conditions to maintain the same structure gives us:

```
int EuclidsGCD(int m, int n)
{
int r;
r = m / n;
if(r != 0)
    {
do
        {
m = n;
r = m % r;
n = r;
}while(r != 0)
    }
return n;
}
```

It is up to the reader to compile this new “optimized” C code, and determine if there is any performance increase. Try compiling this new code without optimizations first, and then with optimizations. Compare the new assembly listings to the previous ones.

Answer 4

The dec/jnz pair operates faster then a loopsz for several reasons. First, dec and jnz pair up in the different modules of the netburst pipeline, so they can be executed simultaneously. Top that off with the fact that dec and jnz both require few cycles to execute, while the loopnz (and all the loop instructions, for that matter) instruction takes more cycles to complete. loop instructions are rarely seen output by good compilers.

Interleaving

Optimizing Compilers will engage in a process called **interleaving** to try and maximize parallelism in pipelined processors. This technique is based on two premises:

1. That certain instructions can be executed out of order and still maintain the correct output
2. That processors can perform certain pairs of tasks simultaneously.

This chapter will talk about code interleaving, and will explore some of the underlying hardware architectures that make interleaving possible, and desirable.

x86 NetBurst Architecture

The Intel **NetBurst Architecture** divides an x86 processor into 2 distinct parts: the supporting hardware, and the primative core processor. The primative core of a processor contains the ability to perform some calculations blindingly fast, but not the instructions that you or I am familiar with. The processor first converts the code instructions into a form called “micro-ops” that are then handled by the primative core processor.

The processor can also be broken down into 4 components, or modules, each of which is capable of performing certain tasks. Since each module can operate separately, up to 4 separate tasks can be handled *simultaneously* by the processor core, so long as those tasks can be performed by each of the 4 modules:

Port0

Double-speed integer arithmetic, floating point load, memory store

Port1

Double-speed integer arithmetic, floating point arithmetic

Port2

memory read

Port3

memory write (writes to address bus)

So for instance, the processor can simultaneously perform 2 integer arithmetic instructions in both Port0 and Port1, so a compiler will frequently go to great lengths to put arithmetic instructions close to each other. If the timing is just right, up to 4 arithmetic instructions can be operating simultaneously.

Notice however that writing to memory is particularly slow (requiring the address to be sent by Port3, and the data itself to be written by Port0). Floating point numbers need to be loaded to the FPU before they can be operated on, so a floating point load and a floating point arithmetic instruction cannot operate on a single value in a single instruction cycle. Therefore, it is not

uncommon to see floating point values loaded, integer values be manipulated, and then the floating point value be operated on.

x86 instruction pairings

Unintuitive Instructions

Optimizing compilers frequently will use instructions that are not intuitive. The fact of the matter is that some instructions can perform unintuitive tasks, often times faster than the more obvious choices.

The only way to know that one instruction is faster then another is to consult the processor documentation. However, knowing some of the most common substitutions is very useful to the reverser.

This chapter will more or less be a table of substitutions, with a few explanatory notes.

Common portable instruction substitutions

Division by a constant can be turned into multiplication by a constant followed by a right shift. Compilers do this all the time.

Common x86 instruction substitutions

lea

The lea instruction has the following form:

```
lea dest, (XS:)[reg1 + reg2 * x]
```

Where XS is a segment register (SS, DS, CS, etc...), reg1 is the base address, reg2 is a variable offset, and x is a multiplicative scaling factor. What lea does, essentially, is load the memory address being pointed to in the second argument, into the first argument. Look at the following example:

```
mov eax, 1
lea ecx, [eax + 4]
```

Now, what is the value of ecx? The answer is that ecx has the value of (eax + 4), which is 5. In essence, lea is used to do addition and subtraction of a register and a constant that is a byte or less (-128 to +127).

Now, consider:

```
mov eax, 1
lea ecx, [eax+eax*2]
```

Now, ecx equals 3.

The difference is that lea is quick (because it only adds a register and a small constant), whereas the **add** and **sub** instructions are more versatile, but slower. lea is used for arithmetic in this fashion very frequently, even when compilers are not actively optimizing the code.

xor

> The xor instruction performs the bit-wise exclusive-or operation on two operands. Consider then, the following example:

```
mov al, 0xAA
xor al, al
```

What does this do? Lets take a look at the binary:

```
10101010 ;10101010 = 0xAA
xor 10101010
```

```
00000000
```

The answer is that “xor reg, reg” sets the register to 0. More importantly, however, is that “xor eax, eax” sets eax to 0 *faster* (and the generated code instruction is smaller) than an equivalent “mov eax, 0”. Trust me.

mov edi, edi

On a 64-bit x86 system, this instruction clears the high 32-bits of the rdi register.

shl, shr

left-shifting, in binary arithmetic, is equivalent to multiplying the operand by 2. Right-shifting is also equivalent to integer division by 2, although the lowest bit is dropped. in general, left-shifting by N spaces multiplies the operand by 2^N, and right shifting by N spaces is the same as dividing by 2^N

xchg

xchg exchanges the contents of two registers, or a register and a memory address. A noteworthy point is the fact that xchg operates faster than a move instruction. For this reason, xchg will be used to move a value from a source to a destination, when the value in the source no longer needs to be saved.

Code Obfuscation

Code Obfuscation is a pretty advanced topic, but we will talk about it here so that a novice reverser won't be completely lost when they are staring at a listing of disassembled garbage. This section *will not* talk about how to obfuscate code, or how to break obfuscation, but instead will only show some basic ways to recognize code that has been obfuscated or encrypted. Further discussion of this topic may be reserved for a later advanced section.

What is Code Obfuscation?

There are many things that obfuscation could be:

- Encrypted code that is decrypted prior to runtime.
- Compressed code that is decompressed prior to runtime.
- Executables that contain Encrypted sections, and a simple decrypter.
- Code instructions that are purposefully put in a hard-to read order (often at the expense of execution speed).

This chapter will try to examine some common methods of obfuscating code, but will not necessarily delve into methods to break the obfuscation.

Section 5: Bytecode

Reversing Bytecode

This section is all about reversing **bytecodes**. Both the Java bytecode, and the MSIL .NET bytecode are considered (and others, if needed).

Java Class Files

.Java Files

Java source files are called .java files, and are written by humans using the ASCII characterset. Compiled Java files however are called .class files, and are what is run by the Java Virtual Machine.

.Class File Structure

The class file format is documented in chapter four of the Java VM specification.

Tools

Documentation for the Java compiler provided by Sun, javac, can be found here. Sun also provides a disassembler for class files, javap. Documentation for javap can be found here.

Other Java Tools

Aside from the Java Compiler and the Java Virtual Machine (both introduced in the chapter: The Java Compiler), there are a number of valuable tools that can be used on java .Class files that reversers might find very interesting indeed. This chapter will discuss some of them.

.NET File Structure

On Windows

On Windows systems, .NET files appear to be normal PE executable files. These “assemblies” contain regular PE header information, and include enough native machine code to direct the flow of execution to the .NET virtual machine. The remainder of the file is the .NET bytecode that gets executed.

On Other Systems

.NET Tools

.NET Framework Tools

MSIL Disassembler (Ildasm.exe)

The MSIL Disassembler is a companion tool to the MSIL Assembler (Ilasm.exe). Ildasm.exe takes a portable executable (PE) file that contains Microsoft intermediate language (MSIL) code and creates a text file suitable as input to Ilasm.exe.

Other Tools

DotNet Reflection

Contains a decompiler, and a powerful object browser.

Lutz Roeder's .NET Reflector

Reflector is a class browser for .NET components. It supports assembly and namespace views, type and member search, XML documentation, call and callee graphs, IL, Visual Basic, Delphi and C# decompiler, dependency trees, base type and derived type hierarchies and resource viewers.

Lattix LDM

LDM reads in .NET code to extract intermodule dependencies which are then used to visualize and manage the architecture of .NET applications. The architecture is represented in a Dependency Structure Matrix (DSM) for a highly scalable representation that allows unwanted dependencies, often a result of unwanted architectural creep, to be identified quickly.

Computer Networks

The chapters in this section are going to talk about **computer networks**, and the reverse-engineering of network communication protocols.

Network Protocols

This section will talk about using packet information to figure out what protocol a certain packet employs.

Protocol Contents

This section will talk about techniques for reverse-engineering the actual contents of a networking protocol. *It might make sense to do some case studies first (below), and then try to make some generalizations from them here.'*

Case Studies

This section will discuss actual case studies of legally reverse-engineered networking protocols

- AIM protocol (for the Gaim project)
- SMB protocol (for the Samba project)
- and more...

Network Architecture

This section will focus on reverse-engineering of a remote network architecture. It will talk about methods to determine just what is behind a remote gateway.

Network Attacks

These pages will deal with common network attacks, such as DoS attacks.

Telnet

Telnet is a great software tool for use in reverse engineering. What telnet is, basically, is a command-line tool for sending and receiving data (both plain text from the console, and binary files) to and from remote servers. Telnet allows the user to specify the address (in IP address form, or as a DNS address), and a port number, in addition to data to be sent after the connection is established.

Packet Sniffers

Packet Sniffers are tools that will read all traffic available on the line, not just information that is addressed to that computer in particular. Packet sniffers can be very useful for identifying traffic on the local network.

Wireshark

One of the most popular packet sniffers and analyser is the open source software package Wireshark (former Ethereal).

Port Scanners

Port scanners are tools that attempt to contact a range of ports (TCP and/or UDP) on a machine or machines, to determine which are open (listening).

The first step in reverse-engineering a network protocol is to identify which port number or numbers it uses. In some cases these are well known, but if not, a port scanner can be used to narrow it down or identify it.

If a goal is simply to understand what services are communicating on a network, or what services are running on a machine, a port scanner is also useful.

NMAP

One widely-used open-source port scanner is NMAP. NMAP is available for multiple platforms, often bundled with Linux distributions, and otherwise available from http://www.insecure.org/nmap/ . Many network services operate on well-known ports, and nmap will not only list open port numbers, but also which service typically uses this port.

Sample output from the command:

```
nmap -sT localhost
```

Shows, among other things, an ssh daemon running on port 22, an smtp mail server running on port 25, and a gnutella file-sharing program running on port 6346:

```
Starting nmap 3.81 ( http://www.insecure.org/nmap/ ) at 2005-12-03 15:18 ES
Interesting ports on localhost (127.0.0.1):
(The 1653 ports scanned but not shown below are in state: closed)
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    open  smtp
111/tcp   open  rpcbind
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
631/tcp   open  ipp
770/tcp   open  cadlock
6346/tcp  open  gnutella
32770/tcp open  sometimes-rpc3
32771/tcp open  sometimes-rpc5
```

It is important to remember that nmap simply associates the port numbers with the service names, it does not verify that it is in fact the named service that is running.

Therefore, one counter-measure against port scanning is to run services on non-standard ports. For example, a ssh daemon typically runs on port 22. A hacker or hacking program trying the easiest exploits might try to ssh to a machine using a common or default username and password combination. Running an ssh daemon on a different port would lead a would-be hacker to believe that the ssh daemon was not running on the machine.

However, this is not very effective as a countermeasure against reverse-engineering a protocol. It is difficult to hide what port a service uses. The best means of protecting a protocol from reverse engineering is to use some form of encryption.

Netstat

Netstat is a useful utility that comes bundled with Windows NT versions. Netstat can be used to show what ports are open, what state the connection is in, the PID of the program using the port, and the target IP address. To get this information from netstat, go into your command prompt (cmp.exe), and type:

```
netstat -no
```

This will list all the information mentioned above. Netstat can also be used for a number of other purposes. To learn what else netstat can do, type:

```
netstat /?
```

Ethernet Headers

Ethernet Headers

When sniffing for packets, it is sometimes helpful (when it is even possible) to examine the raw Ethernet frame. Ethernet headers are arranged as such:

```
|Desination MAC Address|Source MAC Address|Type|---DATA---|CRC|
```

The MAC address fields are both 6 bytes wide, the “Type” field is 2 bytes wide, and the CRC field comprises the last 4 bits of the frame. The Type and the CRC fields can be discardeed (because they are useless to a reverser), but the MAC address fields will give you some information about the source computer and the destination computers. Plus, inside the “DATA” section of the frame is all the rest of the information that is needed. This DATA section will then frequently contain an IP header, a TCP header, and other information, depending on the protocol being used.

TCP, IP Headers

IP Headers

The IP header can contain a large amount of information, such as the IP address of the source and the destination address, the Time-To-Live, and a few other small peices of information. The data field of an IP packet (everything after the header) will contain the meat of what we are looking for.

TCP Headers

A TCP header contains even more information than an Ethernet or IP header. A TCP header contains the ports for both the source computer and the destination computer. Combined with the IP addresses from an IP header, an IP address/TCP port combination can be used to open a connection on either of the machines. TCP headers also include a sequence number, which will tell the reverse engineer how many TCP packets are going to appear in the transmission. If they know how many packets are coming, they can listen for all of them, and intercept a complete transmission.

Network Attacks

Distributed Attacks

Some of the most unfortunate uses of Reverse Engineering are distributed attacks against internet clients or servers. This page will not discuss how to perform these attacks, but we will explain what an attack is, and how they can be defended against.

Open Proxies

Denial of Service Attacks

Ping of Death

> The most basic denial of service attack was created using a vulnerability in the Windows 95 operating system. The attack worked by structuring a network “ping” in such a way that a large amount of said pings could cause a buffer overflow in the Windows network stack, creating the Windows error page called a BSoD (Blue Screen of Death) and crashing the system. Because of this BSoD result, the attack was called a “Ping of Death”. The vulnerability has since been patched and it is no longer possible for a structered ping packet to cause a crash in modern operating systems.

Denial of Service (DoS)

A modern denial of service attack is specially focused to target servers, specifically web servers. Web servers are unique in that they receive certain packets called SYN packets, which tell the web servers that the pages that they serve are being requested over a TCP/IP request. In response, the web server sends back an ACK (acknowledgement) packet that alerts the querying client computer that its request has been received. If the server receives another SYN packet after the ACK packet has been sent, then the web server sends out the page over HTTP to the specified client.

A client computer can exploit this method of communication by sending falsified TCP/IP packets. TCP packets start with a specified amount of data (20 bytes) called a header that contains basic information about the packet including source IP address, TTL (Time to Live), and protocol. By using a raw socket the IP header can be manually crafted so that the source IP address in the header can be replaced with an arbitrary IP address. This allows a client machine to craft fake SYN packets and direct them at the server. These fake packets cannot be blocked by the server since they are indistinguishable from valid page requests, and their IP addresses cannot be blocked because they can be changed at will by the client machines. A server's ability to send out ACK packets is finite, thus a client can sound out fake SYNs, flooding the server and causing a Denial of Service to legitimate viewers of the web site.

Distributed Denial of Service (DDoS)

A Distributed Denial of Service attack is used against larger web sites or web sites with higher quality/quantity of equipment, allowing them to cope with more SYN/ACK transactions than smaller networks. (e.g. Google) In this case, a large number of attackers or numerous computers which the attacker has penetrated and controlled ("bot networks") are all directed to launch DoS attacks on the specified target. This pooling of resources allows the attacker(s) to overcome a previously unassailable target with sheer resources. Although groups of hackers have banded together to attack a single target in the past (YTMND + SomethingAwful vs. eBaum's World) it is much more common for a single hacker to release a virus or trojan horse than secretly infects systems and programs them to become part of a bot net. This type of attack has frequently been used to target the RIAA's website.

Password Cracking

Dictionary Based Attacks

One of the most common (and effective) methods of cracking passwords. The idea is to use every word from the English Dictionary and often alterations of those words (such as appending a number, or reversing the word) as well as slang, acronyms and other common terms.

Brute Force Attacks

Another common method, although arguably slower than a dictionary based attack. A typical password file has anywhere from 10,000 passwords to about 90,000 passwords. To crack a 4 character password, trying a-Z and 0-9, there would be (62^4) 14,776,336 possible combinations. When the length is increased by even 1 (5 character passwords) the amount of possible combinations jumps to 916,132,832.