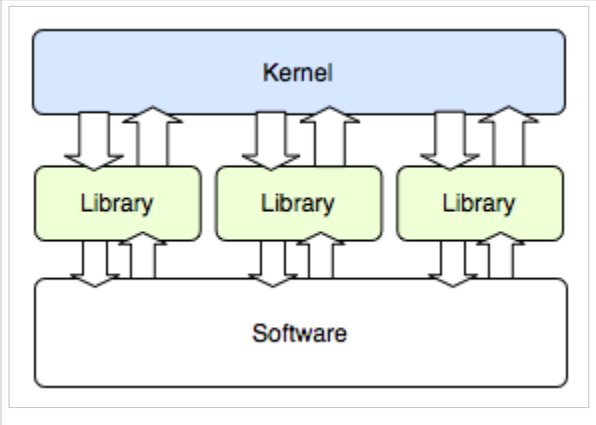
The Wikibook of

x86 Disassembly

Using C and Assembly Language



From Wikibooks: The Free Library

Introduction

What Is This Book About?

This book is about the disassembly of x86 machine code into human-readable assembly, and the decompilation of x86 assembly code into human-readable C or C++ source code. Some topics covered will be common to all computer architectures, not just x86-compatible machines.

What Will This Book Cover?

This book is going to look in-depth at the disassembly and decompilation of x86 machine code and assembly code. We are going to look at the way programs are made using assemblers and compilers, and examine the way that assembly code is made from C or C++ source code. Using this knowledge, we will try to reverse the process. By examining common structures, such as data and control structures, we can find patterns that enable us to disassemble and decompile programs quickly.

Who Is This Book For?

This book is for readers at the undergraduate level with experience programming in x86 Assembly and C or C++. This book is not designed to teach assembly language programming, C or C++ programming, or compiler/assembler theory.

What Are The Prerequisites?

The reader should have a thorough understanding of x86 Assembly, C Programming, and possibly C++ Programming. This book is intended to increase the reader's understanding of the relationship between x86 machine code, x86 Assembly Language, and the C Programming Language. If you are not too familar with these topics, you may want to reread some of the above-mentioned books before continuing.

What is Disassembly?

Computer programs are written originally in a human readable code form, such as assembly language or a high-level language. These programs are then compiled into a binary format called **machine code**. This binary format is not directly readable or understandable by humans. Many programs, such as proprietary commercial programs, or very old legacy programs may not have the source code available to you.

Programs frequently perform tasks that need to be duplicated, or need to be made to interact with other programs. Without the source code and without adequate documentation, these tasks can be difficult to accomplish. This book outlines tools and techniques for attempting to convert the raw machine code of an executable file into equivalent code in assembly language and the high-level languages C and C++. With the high-level code to perform a particular task, several things become possible:

1. Programs can be ported to new computer platforms, by compiling the source code in a different environment.
2. The algorithm used by a program can be determined. This allows other programs to make use of the same algorithm, or for updated versions of a program to be rewritten without needing to track down old copies

of the source code.
3. Security holes and vulnerabilities can be identified and patched by users without needing access to the original source code.
4. New interfaces can be implemented for old programs. New components can be built on top of old components to speed development time and reduce the need to rewrite large volumes of code.

Disassembling code has a large number of practical uses. One of the positive side effects of it is that the reader will gain a better understanding of the relation between machine code, assembly language, and high-level languages. Having a good knowledge of these topics will help programmers to produce code that is more efficient and more secure.

Tools

Assemblers and Compilers

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.

Because assembly is such a simple process, disassembly can often be just as simple. Assembly instructions and machine code words have a one-to-one correspondence, so each machine code word will exactly map to one assembly instruction. However, disassembly has some other difficulties which cannot be accounted for using simple code-word lookups. We will introduce assemblers here, and talk about disassembly later.

Assembler Concepts

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

Assemblers, in general do not perform optimization to the code. The machine 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.*

Some information about the program is lost during the assembly process. First and foremost, program data is stored in the same raw binary format as the machine code instructions. This means that it can be difficult to determine which parts of the program are actually instructions. Notice that you can disassemble raw data, but the resultant assembly code will be nonsensical. Second, textual information from the assembly source code file, such as variable names, label names, and code comments are all destroyed during assembly. When you disassemble the code, the instructions will be the same, but all the other helpful information will be lost. The code will be accurate, but more difficult to read.

Compilers, as we will see later, cause even more information to be lost, and decompiling is often so difficult and convoluted as to become nearly impossible to do accurately.

Intel Syntax Assemblers

Because of the pervasiveness of Intel-based IA-32 microprocessors in the home PC market, the majority of assembly work done (and the majority of assembly work considered in this wikibook) will be x86 based. Many of these assemblers (or new versions of them) can handle IA-64 code as well, although this wikibook 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 isn't 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 its 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 its 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 compatible 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 isn't portable to other systems.

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 its 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 open source 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 its instructions, which some users find to be less readable than 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. [1] (http://www.gnu.org/software/binutils/)

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 outweighed 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.

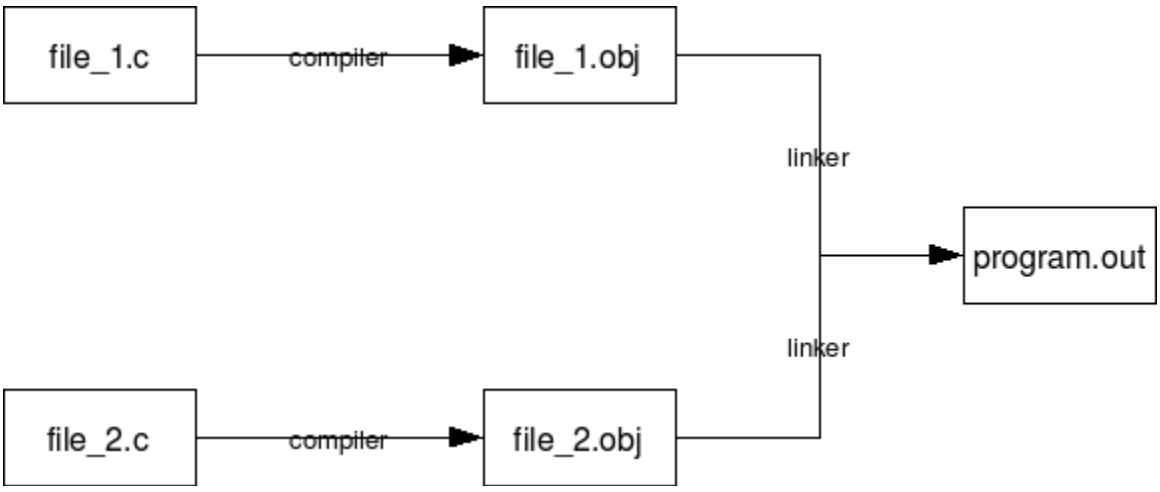
Here is an example of some HLA code:

```
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

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 isn't 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. The figure below shows the common compile-time steps to building a program using the C programming language. The compiler produces object files which are linked to form the final executable:



For the purposes of this book, we will only be considering the case of a compiler that converts C or C++ into assembly code or machine language. Some compilers such as the Microsoft C compiler will compile C and C++ source code directly into machine code. GCC on the other hand will compile C and C++ into assembly language, and an assembler is used to convert that into the appropriate machine code. From the standpoint of a disassembler, it does not matter exactly how the original program was created. Notice also that it is not possible to exactly reproduce the C or C++ code used originally to create an executable. It is, however, possible to create code that compiles identically, or code that performs the same task.

C language statements do not share a one to one relationship with assembly language. Consider that the following C statements will typically all compile into the same assembly language code:

```
*arrayA = arrayB[x++];

*arrayA = arrayB[x]; x++;

arrayA[0] = arrayB[x++];

arrayA[0] = arrayB[x]; x++;
```

Also, consider how the following loop constructs perform identical tasks, and are likely to produce similar or even identical assembly language code:

```
for(;;) { ... }

while(1) { ... }

do { ... } while(1)
```

Common C/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 doesn't need to consider all cases, especially when looking at professional software. This page will discuss each compiler's strengths and weaknesses, its availability (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 doesn't 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 (the .NET bytecode).

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

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

Listing Files

In this wikibook, 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 OSes and architectures. It optimizes well, but has a non-aggressive IA-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 command line 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 IA-32 compatible 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++.

Disassemblers and Decompilers

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. Examples in this book will use Intel and AT&T syntax interchangably. We will typically not use HLA syntax for code examples, but that may change in the future.

x86 Disassemblers

Here we are going to list some commonly available disassembler tools. Notice that there are professional disassemblers (which cost money for a license) and there are freeware/shareware disassemblers. Each disassembler will have different features, so it is up to you as the reader to determine which tools you prefer to use.

Commercial Windows Disassemblers

IDA Pro
: is 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 wikibook will not consider IDA Pro specifically because the price tag is exclusionary. Two freeware versions do exist; see below.
http://www.hex-rays.com/idapro/

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
http://www.heaventools.com/PE_Explorer_disassembler.htm

W32DASM
: W32DASM is an excellent 16/32 bit disassembler for Windows
http://members.cox.net/w32dasm/

Free Windows Disassemblers

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

IDA Pro Freeware 4.3
: Better GUI than the previous version.
http://www.datarescue.be/idafreeware/freeida43.exe

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/

ciasdis
: The official name of ciasdis is computer_intelligence_assembler_disassembler. This Forth-based tool allows to incrementally and interactively build knowledge about a code body. It is unique that all diassembled code can be re-assembled to the exact same code. Processors are 8080, 6809, 8086, 80386, Pentium I en DEC Alpha. A scripting facility aids in analysing Elf and MSDOS headers and makes this tool extendable. The Pentium I ciasdis is available as a binary image, others are in source form, loadable onto lina Forth, available from the same site.
http://home.hccnet.nl/a.w.m.van.der.horst/ciasdis.html

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

As we have alluded to before, there are a number of issues and difficulties associated with the disassembly process. The two most important difficulties are the division between code and data, and the loss of text information.

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 is 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. Some disassemblers (e.g. ciasdis) will allow you to specify rules about whether to disassemble as data or code and invent label names, based on the content of the object under scrutiny. Scripting your own "crawler" in this way is more efficient; for large programs interactive disassembling may be unpractical to the point of being unfeasible.

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). In practice a combination of interactive and automatic analysis and perseverance can handle all but programs specifically designed to thwart reverse engineering, like using encryption and decrypting code just prior to use, and moving code around in memory.

Lost Information

All text-based identifiers, such as variable names, label names, and macros are removed by the assembly process. These identifiers, in addition to comments in the source file, help to make the code more readable to a human, and can also shed some clues on the purpose of the code. Without these comments and identifiers, it is harder to understand the purpose of the source code, and can be difficult to determine the algorithm being used by that code. When you combine this problem with the fact that the code you are trying to read may, in reality, be data (as outlined above), then it can be ever harder to determine what is going on.

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 de-compilation, 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.
http://boomerang.sourceforge.net/

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 than 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

Analysis Tools

Debuggers

Debuggers are programs that allow the user to execute a compiled program one step at at time. You can see what instructions are executed in which order, and which sections of the program are treated as code and which are treated as data. Debuggers allow you to analyze the program while it is running, to help you get a better picture of what it is doing.

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.

mdb
: The Modular Debugger (MDB) is a new general purpose debugging tool for the Solaris™ Operating Environment. Its primary feature is its extensibility. The Solaris Modular Debugger Guide describes how to use MDB to debug complex software systems, with a particular emphasis on the facilities available for debugging the Solaris kernel and associated device drivers and modules. It also includes a complete reference for and discussion of the MDB language syntax, debugger features, and MDB Module Programming API.

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};`

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 free 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, its usefulness is restricted by the fact that it is not free like some of the other options.
http://www.bpsoft.com/

Tiny Hexer
: 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

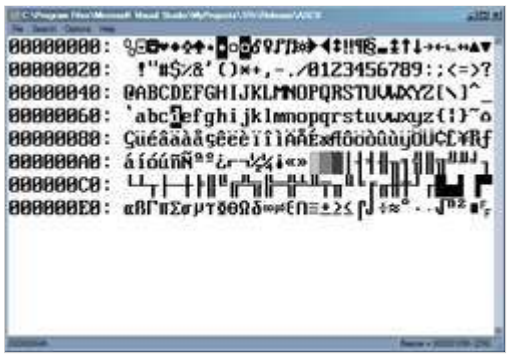
UltraEdit32
: A hex editor/text editor, won "Application of the Year" at 2005 Shareware Industry Awards Conference.
http://www.ultraedit.com/

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/

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/



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

HexEdit
: Powerful and easy to use binary file and disk editor. Free (source available) and shareware versions.
http://www.hexedit.com/

FlexHex
: Provides full support for NTFS files which are based on a more complex model than FAT32 files. Specifically, FlexHex supports Sparse files and Alternate data streams of files on any NTFS volume. Can be used to edit OLE compound files, flash cards, and other types of physical drives.
http://www.heaventools.com/flexhex-hex-editor.htm

Linux Hex Editors

bvi
: a typical three-pane hex editor, with a vi-like interface.

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.

http://home.online.no/~espensa/khexedit/index.html

BIEW
: a viewer of binary files with built-in editor in binary, hexadecimal and disassembler modes. It uses native Intel syntax for disassembly. Highlight AVR/Java/Athlon64/Pentium 4/K7-Athlon disassembler, Russian codepages converter, 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/

DHEX
: DHEX is just another Hexeditor with a Diff-mode for ncurses. It makes heavy use of colors and is themeable.
http://www.dettus.net/dhex/

Hex Editors for Mac

HexEdit
: A simple but reliable hex editor wher you to change highlight colours. There is also a port for Apple Classic users.
http://hexedit.sourceforge.net/

Hex Fiend
: A very simple hex editor, but at least it works fine. It's only 346 KB to download and takes files as big as 116 GB.
http://ridiculousfish.com/hexfiend/

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 **Process Monitor, FileMon, TCPView, RegMon,** and **Process explorer.**
http://www.microsoft.com/technet/sysinternals/default.mspx

API Monitors

SpyStudio Freeware
: The Spy Studio software is a tool to hook into windows processes, log windows API call to DLLs, insert breakpoints and change parameters.
http://www.nektra.com/products/spystudio/

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 wikibook that use dumpbin will call it in this manner.

Here is a list of usefull features of dumpbin [2] (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore/html/_core_dumpbin_options.asp) :

```
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

GNU Tools

The GNU packages have been ported to many platforms including Windows.

GNU BinUtils
: The GNU BinUtils package contains several small utilties that are very useful in dealing with binary files. The most important programs in the list are the GNU objdump, readelf, GAS assembler, and the GNU linker, although the reverser might find more use in addr2line, c++filt, nm, and readelf. http://www.gnu.org/software/binutils/

objdump
: dumps out information about an executable including symbols and assembly. It comes standard. It can be made to support non-native binary formats.

readelf
: like objdump, but more specialized for ELF executables.

size
: lists the sizes of the segments

nm
: lists the symbols in elf file

Other gnu tools

strings
: lists the strings from

file
: tells you what type of file it is

fold
: folds the results of strings into something pageable

GNU Tools for dynamic reverse engineering

kill
: can be used to halt a program - with the sig_stop signal

gdb
: can be used to attach to a program

strace

trace system calls and signals

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

Platforms

Microsoft Windows

Microsoft Windows

The **Windows operating system** is a popular target for reverses for one simple reason: the OS itself (market share, known weaknesses), and most applications for it, are not Open Source or free. Most software on a Windows machine doesn't come bundled with its 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, its 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 (versions 3.1, 3.5, 3.51 and 4.0), Windows 2000 (NT 5.0), Windows XP (NT 5.1), Windows Server 2003 (NT 5.2), and Windows Vista (NT 6.0).

The Microsoft XBOX and and XBOX 360 also run a variant of NT, forked from Windows 2000. 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 action is 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 is much easier to write. Complex operations that involve initializing multiple data structures and calling multiple sub-functions 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 mode is different from 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, each performing 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 than 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 enigmatic 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 real 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; it is rumored that this prefix was chosen due to its having no significance at all.

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 Windows 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 than 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 user mode, 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 released a new version of its Windows operation system, named "Windows Vista." Windows Vista may be better known by its 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 Vista was released January 30th, 2007.

Windows CE/Mobile, 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.

"Non-Executable 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. Although COM provides cross-platform (to some extent) and cross-language facilities, each COM object is compiled to a native binary, rather than an intermediate format such as Java or .NET. As a result, COM does not require a virtual machine to execute such objects.

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 (http://en.wikipedia.org/wiki/Active_Template_Library) 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)

RPC is a generic term referring to techniques that allow a program running on one machine to make calls that actually execute on another machine. Typically, this is done by *marshalling* all of the data needed for the procedure including any state information stored on the first machine, and building it into a single data structure, which is then transmitted over some communications method to a second machine. This second machine then performs the requested action, and returns a data packet containing any results and potentially changed state information to the originating machine.

In Windows NT, RPC is typically handled by having two libraries that are similarly named, one which generates RPC requests and accepts RPC returns, as requested by a user-mode program, and one which responds to RPC requests and returns results via RPC. A classic example is the print spooler, which consists of two pieces: the RPC stub spoolss.dll, and the spooler proper and RPC service provider, spoolsv.exe. In most machines, which are stand-alone, it would seem that the use of two modules communicating by means of RPC is overkill; why not simply roll them into a single routine? In networked printing, though, this makes sense, as the RPC service provider can be resident physically on a distant machine, with the remote printer, and the local machine can control the printer on the remote machine in exactly the same way that it controls printers on the local machine.

Windows Executable 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.

Notice that MS-DOS COM files (short for "command" files) are not the same as *Component-Object Model* files, which are an object-oriented library technology.

MS-DOS EXE Files

One way MS-DOS compilers got around the 64k memory limitation was with the introduction of **memory models**. The basic concept is to cleverly set different segment registers in the x86 CPU (CS, DS, ES, SS) to point to the same or different segments, thus allowing varying degrees of access to memory. Typical memory models were:

tiny
: All memory access are 16-bit (never reload any segment register). Produces a .COM file instead of an .EXE file.

small
: All memory access are 16-bit (never reload any segment register).

compact
: accesses to the code segment reload the CS register, allowing 32-bit of code. Data accesses don't reload the DS, ES, SS registers, allowing 16-bit of data.

medium
: accesses to the data segment reload the DS, ES, SS register, allowing 32-bit of data. Code accesses don't reload the CS register, allowing 16-bit of code.

large
: both code and data accesses use the segment registers (CS for code, DS, ES, SS for data), allowing 32-bit of code and 32-bit of data.

huge
: same as the large model, with additional arithmetic being generated by the compiler to allow access to arrays larger than 64k.

When looking at a COM file, one has to decide which memory model was used to build that file.

PE Files

Portable Executable file is the standard binary(EXE and DLL) file format on Windows NT, Windows 95 and Win32. The Win32 SDK has a file `winnt.h`, which declares various structs and variables used in the PE files. A DLL, imagehlp.dll also contains some functions for manipulating PE files. PE files are broken down into various sections that can be examined.

Relative Virtual Addressing (RVA)

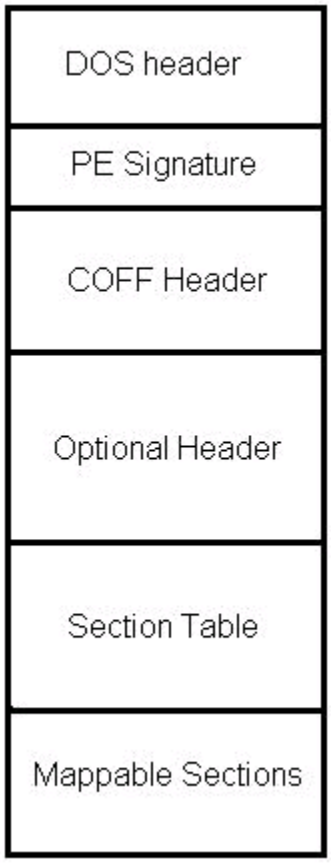
In a Windows environment, executable modules can be loaded at any point in memory, and are expected to run without problem. To allow multiple programs to be loaded at seemingly random locations in memory, PE files have adopted a tool called RVA: Relative Virtual Addresses. RVA's assume that the "base 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".

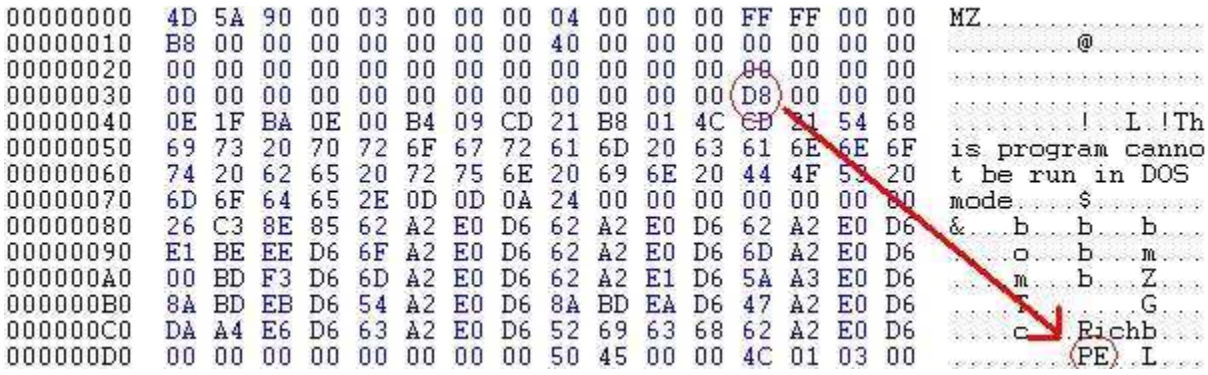
The DOS header is also known by some as the EXE header. Here is the DOS header presented as a C data structure:

```
struct DOS_Header
 {
     char signature[2] = "MZ";
     short lastsize;
     short nblocks;
     short nreloc;
     short hdrsize;
     short minalloc;
     short maxalloc;
     void *ss;
     void *sp;
     short checksum;
     void *ip;
     void *cs;
     short relocpos;
     short noverlay;
     short reserved1[4];
     short oem_id;
     short oem_info;
     short reserved2[10];
     long  e_lfanew;
 }
```

Immediately following the DOS Header will be the classic error message "This program cannot be run in DOS mode".

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 chunk 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 wikibook 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 NumberOf
 }                                        //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 exist 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. .testbss/TEXTBSS - Present if Incremental Linking is enabled
3. .data/.idata/DATA/IDATA - Contains initialised data
4. .bss/BSS - Contains uninitialised data

Section Flags

The section flags is a 32-bit 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
#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 th
#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 its 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 the 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 arrays of RVA's, each of which point to a IMAGE_IMPORT_BY_NAMES struct. The arrays are terminated with an entry that is less than or equal to zero. These two arrays are parallel 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 parallel 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 parallel 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 its 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 dialog boxes, icons, menus, images, and other types of resources, including arbitrary binary data. 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 portion 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 also has a dual purpose: if the most significant bit is set, the remaining 31 bits form an offset from the start of the resource data to another IMAGE_RESOURCE_DIRECTORY (i.e. this entry is an interior node of the resource tree). Otherwise, this is a leaf node, and Data contains the offset from the start of the resource data to a structure which describes the specifics of the resource data itself (which can be considered to be an ordered stream of bytes):

```
struct IMAGE_RESOURCE_DATA_ENTRY
{
        long *Data;
        long Size;
        long CodePage;
        long Reserved;
}
```

The Data value contains an RVA to the actual resource data, Size is self-explanatory, and CodePage contains the Unicode codepage to be used for decoding Unicode-encoded strings in the resource (if any). Reserved should be set to 0.

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 the resource: 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, the format of which is not defined by the PE specification and can be treated as an arbitrary stream of bytes.

Relocations

Alternate Bound Import Structure

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>
```

Linux

The Print Version page of the x86 Disassembly Wikibook is a stub. You can help by expanding this section.

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 its 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 (http://www.die.net/doc/linux/man/man1/gdb.1.html)

winedbg
: A debugger for Wine, used to debug Win32 executables under Linux. manpage (http://www.die.net/doc/linux/man/man1/winedbg.1.html)

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 (http://www.doc.ic.ac.uk/lab/labman/lookup-man.cgi?strings(1))

file
: Determines a file type, useful for determining whether an executable has been stripped and whether it's been dynamically (or statically) linked. manpage (http://www.doc.ic.ac.uk/lab/labman/lookup-man.cgi?file(1))

Linux Executable Files

The Print Version page of the x86 Disassembly Wikibook is a stub. You can help by expanding this section.

a.out Files

a.out is a very simple format consisting of a header (at offset 0) which contains the size of 3 executable sections (code, data, bss), plus pointers to additional information such as relocations (for .o files), symbols and symbols' strings. The actual sections contents follows the header. Offsets of different sections are computed from the size of the previous section.

File Format

ELF Files

The **ELF file format** (short for Executable and Linking Format) was developed by 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 than 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.

ELF Objects are broken down into different segments and/or sections. These can be located by using the ELF header found at the first byte of the object. The ELF header provides the location for both the program header and the section header. Using these data structures the rest of the ELF objects contents can be found, this includes .text and .data segments which contain code and data respectively.

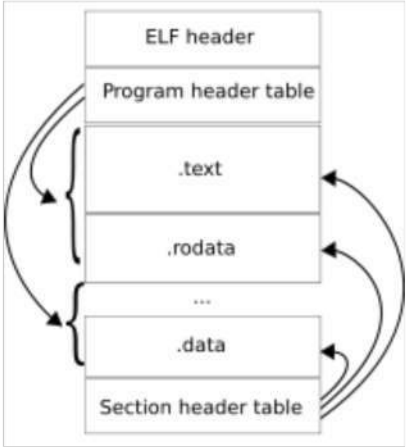
The GNU readelf utility, from the binutils package, is a common tool for parsing ELF objects.

File Format

Each ELF file is made up of one ELF header, followed by file data. The file data can include:

- Program header table, describing zero or more segments
- Section header table, describing zero or more sections
- Data referred to by entries in the program or section header table

The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes, which are not covered by a section. In the normal case of a Unix executable one or more sections are enclosed in one segment.



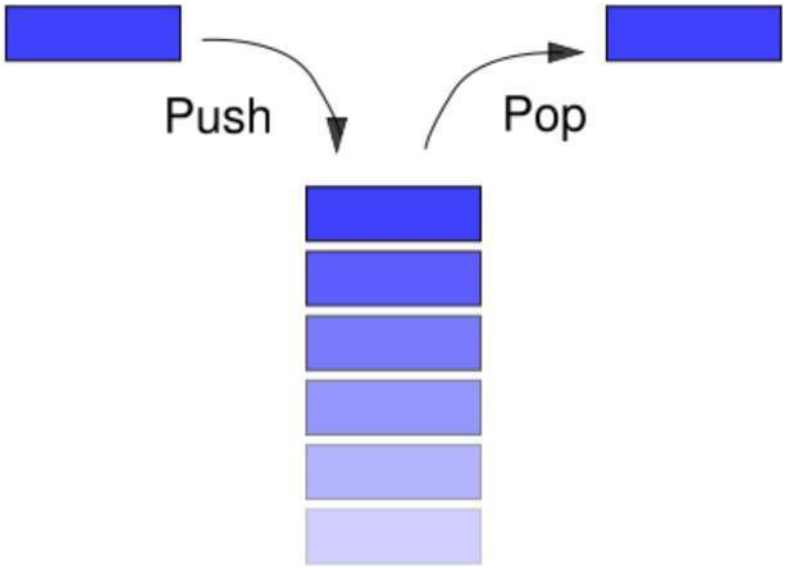
An ELF file has two views: the program header shows the *segments* used at run-time, while the section header lists the set of *sections* of the binary.

Relocatable ELF Files

Code Patterns

The Stack

The Stack



Generally speaking, a **stack** is a data structure that stores data values contiguously in memory. Unlike an array, however, you access (read or write) data only at the "top" of the stack. To read from the stack is said "**to pop**" and to write to the stack is said "**to push**". A stack is also known as a LIFO queue (Last In First Out) since values are popped from the stack in *the reverse order* that they are pushed onto it (think of how you pile up plates on a table). Popped data disappears from the stack.

All x86 architectures use a stack as a temporary storage area in RAM that allows the processor to quickly store and retrieve data in memory. The current top of the stack is pointed to by the **esp** register. The stack "grows" downward, from high to low memory addresses, so values recently pushed onto the stack are located in memory addresses *above* the esp pointer. No register specifically points to the bottom of the stack, although most operating systems monitor the stack bounds to detect both "underflow" (popping an empty stack) and "overflow" (pushing too much information on the stack) conditions.

When a value is popped off the stack, the value remains sitting in memory until overwritten. However, you should never rely on the content of memory addresses below esp, because other functions may overwrite these values without your knowledge.

Users of Windows ME, 98, 95, 3.1 (and earlier) may fondly remember the infamous "Blue Screen of Death" that was usually caused by a stack overflow exception. This occurs when too much data is written to the stack, and the stack "grows" beyond its limits. Modern operating systems use better bounds-checking and error recovery to reduce the occurrence of stack overflows, and to maintain system stability after one has occurred.

Push and Pop

The following lines of ASM code are basically equivalent:

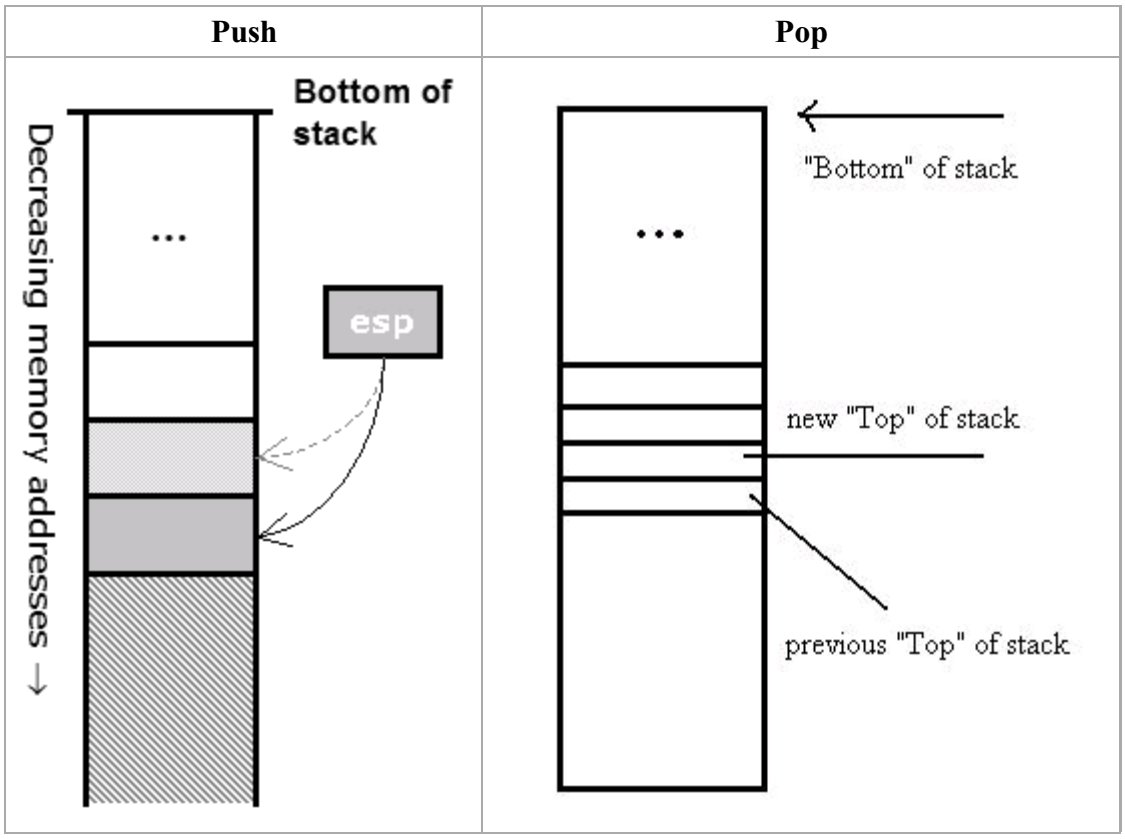
```
push eax
```

```
sub esp, 4
mov DWORD PTR SS:[esp], eax
```

```
pop eax
```

```
mov eax, DWORD PTR SS:[esp]
add esp, 4
```

but the single command actually performs much faster than the alternative.



ESP In Action

Let's say we want to quickly discard 3 items we pushed earlier onto the stack, without saving the values (in other words "clean" the stack). The following works:

This code example uses **MASM Syntax**

```
pop eax
pop eax
pop eax
xor eax, eax  ; clear eax because we don't need the values
```

However there is a faster method. We can simply perform some basic arithmetic on esp to make the pointer go "above" the data items, so they cannot be read anymore, and can be overwritten with the next round of **push** commands.

```
add esp, 12  ; 12 is 3 DWORDs (4 bytes * 3)
```

Likewise, if we want to reserve room on the stack for an item bigger then a DWORD, we can use a subtraction to artificially move esp forward. We can then access our reserved memory directly as a memory pointer, or we can access it indirectly as an offset value from esp itself.

Say we wanted to create an array of byte values on the stack, 100 items long. We want to store the pointer to the base of this array in **edi**. How do we do it? Here is an example:

```
sub esp, 100  ; num of bytes in our array
mov edi, esp  ; copy address of 100 bytes area to edi
```

To destroy that array, we simply write the instruction

```
add esp, 100
```

Reading Without Popping

To read values on the stack without popping them off the stack, **esp** can be used with an offset. For instance, to read the 3 DWORD values from the top of the stack into eax (but without using a pop instruction), we would use the instructions:

```
mov eax, DWORD PTR SS:[esp]
mov eax, DWORD PTR SS:[esp + 4]
mov eax, DWORD PTR SS:[esp + 8]
```

Remember, since esp moves downward as the stack grows, data on the stack can be accessed with a positive offset. A negative offset should never be used because data "above" the stack cannot be counted on to stay the way you left it. The operation of reading from the stack without popping is often referred to as "peeking", but since this isn't the official term for it this wikibook won't use it.

Data Allocation

There are two areas in the computer memory where a program can store data. The first that we have been talking about is the stack. It is a linear LIFO buffer that allows fast allocations and deallocations, but has a limited size. The **heap** is typically a non-linear data storage area, typically implemented using linked lists, binary trees, or other more exotic methods. Heaps are slightly more difficult to interface with and to maintain then a stack, and allocations/deallocations are performed more slowly. However, heaps can grow as the data grows, and new heaps can be allocated when data quantities become too large.

As we shall see, explicitly declared variables are allocated on the stack. Stack variables are finite in number, and have a definite size. Heap variables can be variable in number and in size. We will discuss these topics in more detail later.

Functions and Stack Frames

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.

When a function is called, a new stack frame is created at the current **esp** location. A stack frame acts like a partition on the stack. All items from previous functions are higher up on the stack, and should not be modified. Each current function has access to the remainder of the stack, from the stack frame until the end of the stack page. The current function always has access to the "top" of the stack, and so functions do not need to take account of the memory usage of other functions or programs.

Standard Entry Sequence

For many compilers, the standard function entry sequence is the following piece of code (*X* is the total size, in bytes, of all *automatic* variables used in the function):

This code example uses **MASM Syntax**

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

For example, here is a C function code fragment and the resulting assembly instructions:

```
void MyFunction()
{
  int a, b, c;
  ...
```

```
push ebp     ; save the value of ebp
mov ebp, esp ; ebp now points to the top of the stack
sub esp, 12  ; space allocated on the stack for the local variables
```

This means local variables can be accessed by referencing ebp. Consider the following C code fragment and corresponding assembly code:

```
a = 10;
b = 5;
c = 2;
```

```
mov [ebp -  4], 10  ; location of variable a
mov [ebp -  8], 5   ; location of b
mov [ebp - 12], 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 then 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)
{
  ...
}
```

It 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
```

Note: Remember that the **call** x86 instruction is basically equivalent to

```
push eip + 2 ; return address is current address + size of two instructions
jmp _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. As the function contents pushes and pops values, ebp is not affected by esp. Remember that pushing basically does this:

```
sub esp, 4   ; "allocate" space for the new stack item
mov [esp], X ; put new stack item value X in
```

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 (crude) representation of the stack at this point:

```
:     :
|  5  | [ebp + 12] (2nd function argument)
| 10  | [ebp + 8]  (1st function argument)
| RA  | [ebp + 4]  (return address)
| FP  | [ebp]      (old ebp value)
|     | [ebp - 4]  (1st local variable)
:     :
```

The stack pointer value may change during the execution of the current function. In particular this happens when:

- parameters are passed to another function;
- the pseudo-function "alloca()" is used.

This means that the value of **esp** cannot be reliably used to determine (using the appropriate offset) the memory location of a specific local variable. To solve this problem, many compilers access local variables using negative offsets from the **ebp** registers. This allows us to assume that the same offset is always used to access the same variable (or parameter). For this reason, the ebp register is called the **frame pointer**, or FP.

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)
;x = [ebp + 16], y = [ebp + 12], z = [ebp + 8]
;a = [ebp - 12] = [esp], b = [ebp - 8] = [esp + 4], c = [ebp - 4] = [esp + 8]
mov esp, ebp
pop ebp
ret
```

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 that is only referenced by calls (no referenced through function pointers), 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. We'll discuss local static variables and other types of variables in a later chapter.

Functions and Stack Frame Examples

Example: Number of Parameters

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?

This code example uses **MASM Syntax**

```
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
```

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;
 }
```

Example: Standard Entry Sequences

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

This code example uses **MASM Syntax**

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

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

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.

Calling conventions specify how arguments are passed to a function, how return values are passed back out of a function, how the function is called, and how the function manages the stack and it's stack frame. In short, the calling convention specifies how a function call in C or C++ is converted into assembly language. Needless to say, there are many ways for this translation to occur, which is why it's so important to specify certain standard methods. If these standard conventions did not exist, it would be nearly impossible for programs created using different compilers to communicate and interact with one another.

There are three major calling conventions that are used with the C language: STDCALL, CDECL, and FASTCALL. In addition, there is another calling convention typically used with C++: THISCALL. There are other calling conventions as well, including PASCAL and FORTRAN conventions, among others. We will not consider those conventions in this book.

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 its 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 documentation about implementation specifics.

If a calling convention other than CDECL is to be used, or if CDECL is not the 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* (aka variadic functions). 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.

Variadic functions usually have special entry code, generated by the va_start(), va_arg() C pseudo-functions.

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.)
- 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 eax, 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.

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 isn't 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 **name mangling**.

Name Mangling

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. Notice that in many cases, it's possible to determine which compiler created the executable by examining the specifics of the name-mangling format. We will not cover this topic in this much depth in this book, however.

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
```

Extern "C"

In a C++ source file, functions placed in an `extern "C"` block are guaranteed not to be mangled. This is done frequently when libraries are written in C, and the functions need to be exported without being mangled. Even though the program is written in C++ and compiled with a C++ compiler, some of the functions might therefore not be mangled and will use one of the ordinary C calling conventions (typically CDECL).

Note on Name Decorations

We've been discussing name decorations in this chapter, but the fact is that in pure disassembled code there typically are no names whatsoever, especially not names with fancy decorations. The assembly stage removes all these readable identifiers, and replaces them with the binary locations instead. Function names really only appear in two places:

1. Listing files produced during compilation
2. In export tables, if functions are exported

When disassembling raw machine code, there will be no function names and no name decorations to examine. For this reason, you will need to pay more attention to the way parameters are passed, the way the stack is cleaned, and other similar details.

While we haven't covered optimizations yet, suffice it to say that optimizing compilers can even make a mess out of these details. Functions which are not exported do not necessarily need to maintain standard interfaces, and if it is determined that a particular function does not need to follow a standard convention, some of the details will be optimized away. In these cases, it can be difficult to determine what calling conventions were used (if any), and it is even difficult to determine where a function begins and ends. This book cannot account for all possibilities, so we try to show as much information as possible, with the knowledge that much of the information provided here will not be available in a true disassembly situation.

Calling Convention Examples

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 than 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 ridiculous 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.

Example: C Calling Conventions

Identify the calling convention of the following C function:

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

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

Example: Named Assembly Function

Identify the calling convention of the function **MyFunction**:

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

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

Example: Unnamed Assembly Function

This code snippet is the entire body of an unnamed assembly function. Identify the calling convention of this function.

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

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.

Example: Another Unnamed Assembly Function

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

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.

Example: Name Mangling

What can we tell about the following function call?

```
move ecx, x
 push eax
 mov eax, ss:[ebp - 4]
 push eax
 mov al, ss:[ebp - 3]
 call @__Load?$Container__XXXY_?Fcii
```

Two things should get our attention immediately. The first is that before the function call, a value is stored into ecx. Also, the function name itself is heavily mangled. This example must use the C++ THISCALL convention. Inside the mangled name of the function, we can pick out two english words, "Load" and "Container". Without knowing the specifics of this name mangling scheme, it is not possible to determine which word is the function name, and which word is the class name.

We can pick out two 32-bit variables being passed to the function, and a single 8-bit variable. The first is located in **eax**, the second is originally located on the stack from offset -4 from **ebp**, and the third is located at **ebp** offset -3. In C++, these would likely correspond to two **int** variables, and a single **char** variable. Notice at the end of the mangled function name are three lower-case characters "cii". We can't know for certain, but it appears these three letters correspond to the three parameters (char, int, int). We do not know from this whether the function returns a value or not, so we will assume the function returns **void**.

Assuming that "Load" is the function name and "Container" is the class name (it could just as easily be the other way around), here is our function definition:

```
class Container
{
  void Load(char, int, int);
}
```

Branches

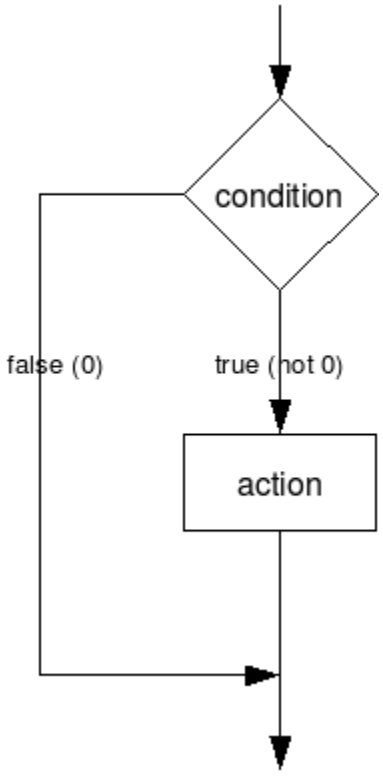
Branching

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(condition)
{
  action;
}
```



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 condition goto end
  action
end:
```

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)
{
   x++;
}
```

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:
...
```

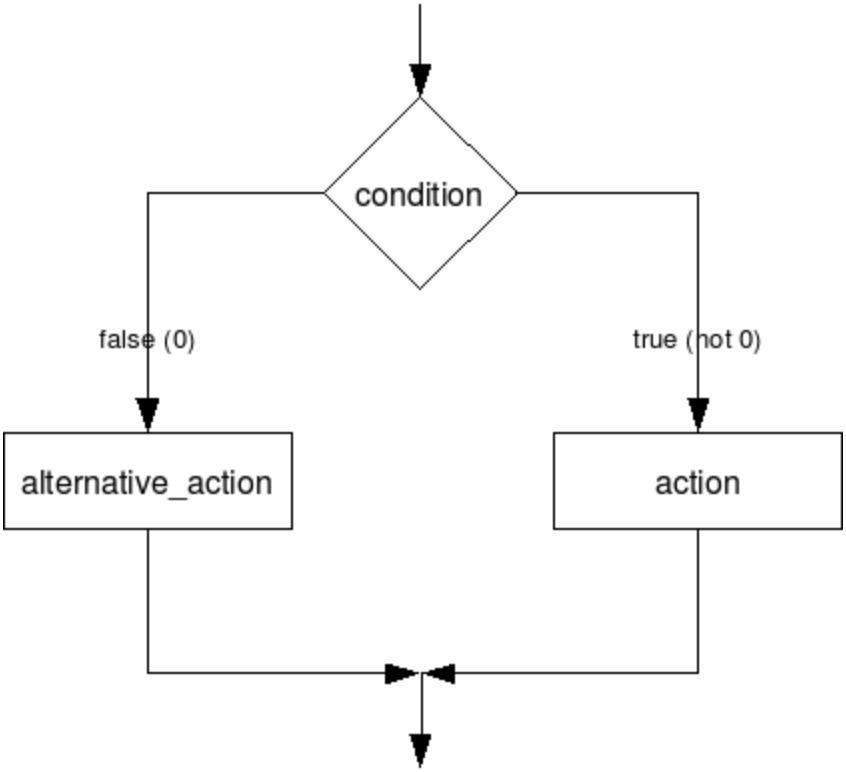
X is incremented in the c code only if it is **less than** y, so the assembler code now jumps if it's greater than or

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(condition)
 {
   action;
 }
 else
 {
   alternative_action;
 }
```



Now, what happens here? Like before, the if statement only jumps to the else clause when *condition* is false. However, we must also install an *unconditional* jump at the end of the "then" clause, so we don't perform the else clause directly afterwards.

Here is the above example in pseudocode:

```
if not condition goto else
  action
  goto end
else:
  alternative_action
end:
```

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 didn't have any local variables, however, so perhaps the compiler just needed some extra scratch space to store intermediate values. Why doesn't 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 in-depth 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]
```

This sequence performs the following pseudo-C operation:

```
if( argc - 1 >= 4 )
{
   goto $L810;   /* the default */
}
label *L818[] = { $L806, $L807, $L808, $L809 };  /* define a table of jumps, one per each case */
//
goto L818[argc - 1];   /* use the address from the table to jump to the correct case */
```

Here's why...

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 don't 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. Let's 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, $L818 *isn't the offset, it's the base*, edx*4 is the offset. As we said earlier, edx contains the value of (argc - 1). If argc == 1, we jump to [$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 exit 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. Isn't that the exact opposite of what we wanted? In this case, it is. Let's look at 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 than 1), the **setle** instruction doesn't 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 **two's-complement** format. For instance, let's 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 than 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 than cl.exe produces. However, the stack frame is set up the same way. Notice also that GCC doesn't 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 (because lea (edx,edx,4) means edx + edx * 4, i.e. edx * 5).

Branch Examples

Example: Number of Parameters

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.

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

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);
```

Example: Identify Branch Structures

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?

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

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 cmp 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++;
 }
```

Example: Convert To C

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

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

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 isn't a particularly useful function, but at least we know what it does.

Loops

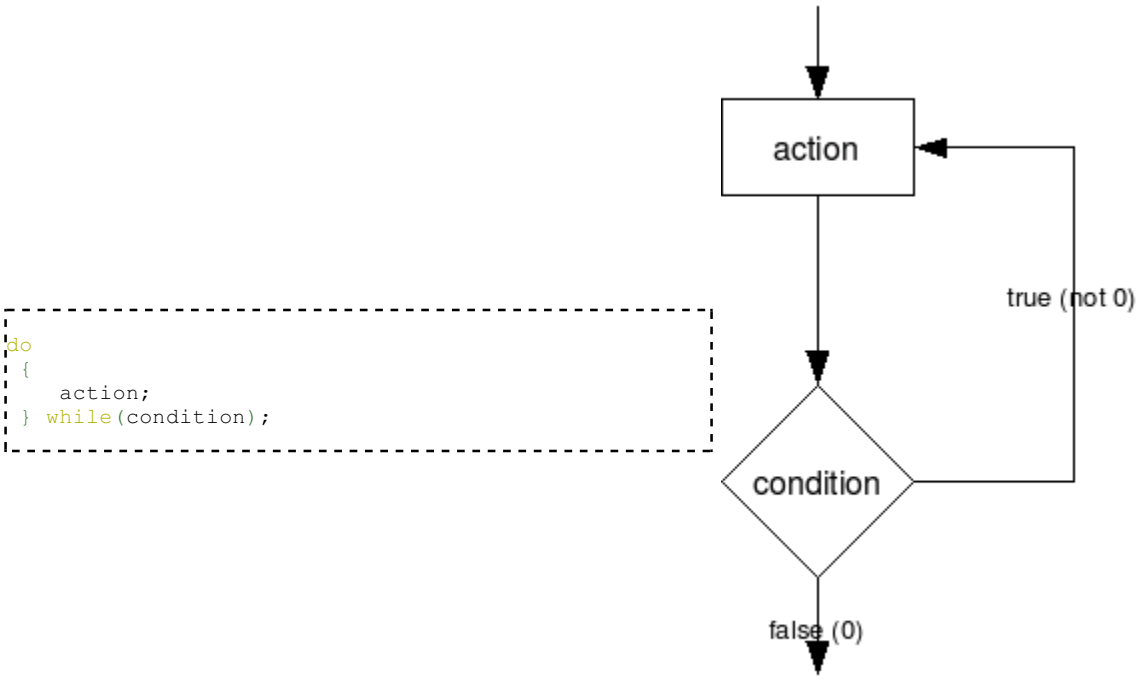
Loops

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
 {
    action;
 } while(condition);
```

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:

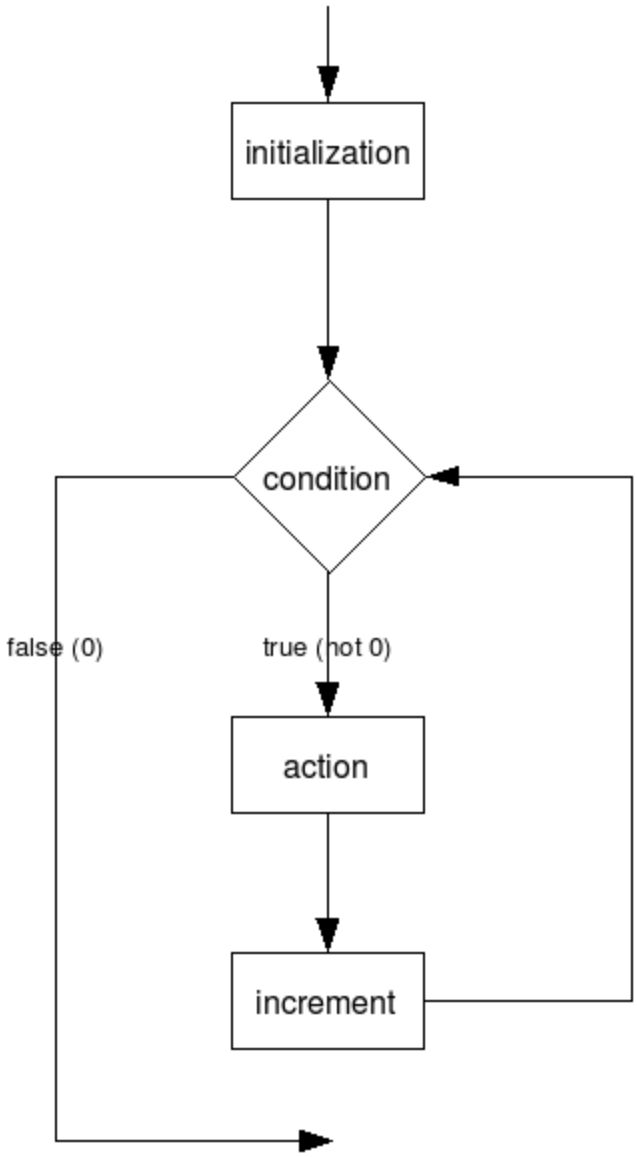
```
if(x <= 10) //remember: in If statements, we reverse the condition from the asm
 {
   do
   {
     x++;
   } while(x <= 10)
 }
```

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(initialization; condition; increment)
 {
   action
 }
```



gets translated into the following pseudocode while-loop:

```
initialization;
 while(condition)
 {
   action;
   increment;
 }
```

Which in turn gets translated into the following Do-While Loop:

```
initialization;
 if(condition)
 {
    do
    {
       action;
       increment;
    } while(condition);
 }
```

Note that often in for() loops you assign an initial constant value in A (for example x = 0), and then compare that value with another constant in B (for example x < 10). Most optimizing compilers will be able to notice that the first time x IS less than 10, and therefore there is no need for the initial if(B) statement. In such cases, the compiler will simply generate the following sequence:

```
initialization;
 do
 {
    action
    increment;
 } while(condition);
```

rendering the code indistinguishable from a while() loop.

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
```

Notice that some non-optimizing compilers will produce nonsensical code for this:

```
mov ax, 1
 cmp ax, 1
 jne loopend
 beginning:
 ;loop body
 cmp ax, 1
 je beginning
 loopend:
```

Notice that a lot of the comparisons here are not needed since the condition is a constant. Most compilers will optimize cases like this.

Loop Examples

Example: Identify Purpose

What does this function do? What kinds of parameters does it take, and what kind of results (if any) does it return?

```
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
 jne _Label_1
 mov esp, ebp
 pop ebp
 ret 4
```

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.

Example: Complete C Prototype

What is this functions C prototype? Make sure to include parameters, return values, and calling convention.

```
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
 jne _Label_1
 mov esp, ebp
 pop ebp
 ret 4
```

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);
```

Example: Decompile To C Code

Decompile this code into equivalent C source code.

```
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
 jne _Label_1
 mov esp, ebp
 pop ebp
 ret 4
```

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;
 }
```

Data Patterns

Variables

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]
```

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

Both .bss and .data sections contain values which can change at run-time (e.g. *variables*). Typically, variables that are initialized to a non-zero value in the source are allocated in the .data section (e.g. "int a = 10;"). Variables that are not initialized, or initialized with a zero value, can be allocated to the .bss section (e.g. "int

arr[100];"). Because all values of .bss variables are guaranteed to be zero at the start of the program, there is no need for the linker to allocate space in the binary file. Therefore, .bss sections do not take space in the binary file, regardless of their size.

"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 wikibook, 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.

Signed and Unsigned Variables

Integer formatted variables, such as **int**, **char**, **short** and **long** may be declared signed or unsigned variables in the C source code. There are two differences in how these variables are treated:

1. Signed variables use signed instructions such as **add**, and **sub**. Unsigned variables use unsigned arithmetic instructions such as **addi**, and **subi**.
2. Signed variables use signed branch instructions such as **jge** and **jl**. Unsigned variables use unsigned branch instructions such as **jae**, and **jb**.

The difference between signed and unsigned instructions is the conditions under which the various flags for greater-then or less-then (overflow flags) are set. The integer result values are exactly the same for both signed and unsigned data.

Floating-Point Values

Floating point values tend to be 32-bit data values (for **float**) or 64-bit data values (for **double**). These values are distinguished from ordinary integer-valued variables because they are used with floating-point instructions. Floating point instructions typically start with the letter *f*. For instance, **fadd**, **fcmp**, and similar instructions are used with floating point values. Of particular note are the **fload** instruction and variants. These instructions take an integer-valued variable and converts it into a floating point variable.

We will discuss floating point variables in more detail in a later chapter.

Global Variables

Global variables do not have a limited scope like lexical variables do inside a function body. Since the notion of lexical scope implies the use of the system stack, and since global variables are not lexical in nature, they are typically not found on the stack. Global variables tend to exist in the program as a hard-coded memory address, a location which never changes throughout program execution. These could exist in the DATA segment of the executable, or anywhere else that a hard-coded memory address can be used to store data.

In C, global variables are defined outside the body of any function. There is no "global" keyword. Any variable which is not defined inside a function is global. In C however, a variable which is not defined inside a function is only global to the particular source code file in which it is defined. For example, we have two files `Foo.c` and `Bar.c`, and a global variable `MyGlobalVar`:

Foo.c

```
int MyGlobalVar;

int GetVarFoo(void)
{
  return MyGlobalVar; //right!
}
```

Bar.c

```
int GetVarBar(void)
{
  return MyGlobalVar; //wrong!
}
```

In the example above, the variable `MyGlobalVar` is visible inside the file `Foo.c`, but is not visible inside the file `Bar.c`. To make `MyGlobalVar` visible inside all project files, we need to use the `extern` keyword, which we will discuss below.

"`static`" Variables

The C programming language specifies a special keyword "`static`" to define variables which are lexical to the function (they cannot be referenced from outside the function) but they maintain their values across function calls. Unlike ordinary lexical variables which are created on the stack when the function is entered and are destroyed from the stack when the function returns, static variables are created once and are never destroyed.

```
int MyFunction(void)
{
  static int x;
  ...
}
```

Static variables in C are global variables, except the compiler takes precautions to prevent the variable from being accessed outside of the parent function's scope. Like global variables, static variables are referenced using a hardcoded memory address, not a location on the stack like ordinary variables. However unlike globals, static variables are only used inside a single function. There is no difference between a global variable which is only used in a single function, and a static variable inside that same function. However, it's good programming practice to limit the number of global variables, so when disassembling, you should prefer interpreting these variables as static instead of global.

"`extern`" Variables

The `extern` keyword is used by a C compiler to indicate that a particular variable is global to the entire project, not just to a single source code file. Besides this distinction, and the slightly larger lexical scope of extern variables, they should be treated like ordinary global variables.

In static libraries, variables marked as being extern might be available for use with programs which are linked to the library.

Global Variables Summary

Here is a table to summarize some points about global variables:

	How it's referenced	Lexical scope	Notes
`static` variables	Hard-coded memory address, only in one function	One function only	In disassembly, indistinguishable from global variables except that it's only used in one function. A global variable is only static if it's never used in another function.
Global variables	Hard-coded memory address, only in one file	One source code file only	Global variables are only used in a single file. This can help you when disassembling to get a rough estimate for how the original source code was arranged.
`extern` variables	Hard-coded memory address, in the entire project	The entire project	Extern variables are available for use in all functions of a project, and in programs linked to the project (external libraries, for example).

When disassembling, a hard-coded memory address should be considered to be an ordinary global variable unless you can determine from the scope of the variable that it is static or extern.

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. For this reasons, some compilers may chose to store constant variables (especially strings) in the .text section of the executable, thus allowing the sharing of these variables across multiple instances of the same process. This creates a big problem for the reverser, who now has to decide whether the code he's looking at is part of a constant variable or part of a subroutine.

"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
```

Variable Examples

Example: Identify C++ Code

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
```

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.

Example: Identify C++ Code

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
```

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).

Data Structures

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 the 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 pseudo-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 pseudo-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[4]? what about buffer[5]? what about buffer[8]? 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.

Spotting 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.

Spotting 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) and 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*.

Since most times an array is accessed through a computed index, not through a constant, the compiler will likely use the following to access an element of the array:

```
mov [ebx + eax], 0x00
```

If the array holds elements larger than 1 byte (for char), the index will need to be multiplied by the size of the element, yielding code similar to the following:

```
mov [ebx + eax * 4], 0x11223344   # access to an array of DWORDs, e.g.    arr[i] = 0x11223344
...
mul eax, $20                       # access to an array of structs, each 20 bytes long
lea edi, [ebx + eax]               # e.g.     ptr = &arr[i]
```

This pattern can be used to distinguish between accesses to arrays and accesses to structure data members.

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 its 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
```

```
;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 is 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 its 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 one 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
   {
      ...
```

```
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 wikibook will not spend too much time on this subject.

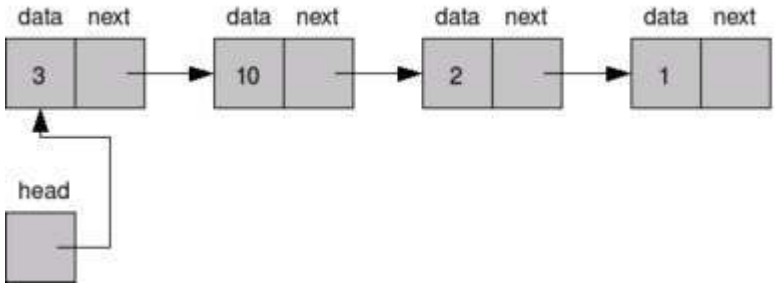
Identifying Structs and Arrays

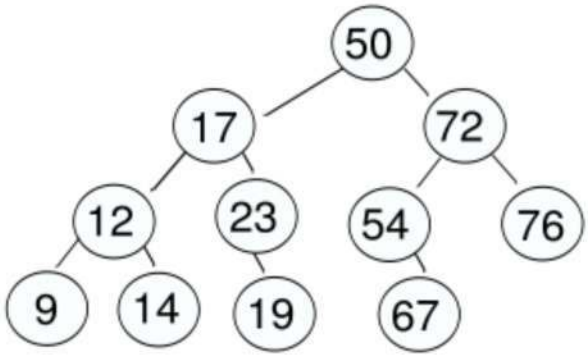
Array elements and structure fields are both accessed as offsets from the array/structure pointer. When disassembling, how do we tell these data structures apart? Here are some pointers:

1. array elements are not meant to be accessed individually. Array elements are typically accessed using a variable offset
2. Arrays are frequently accessed in a loop. Because arrays typically hold a series of similar data items, the best way to access them all is usually a loop. Specifically, `for(x = 0; x < length_of_array; x++)` style loops are often used to access arrays, although there can be others.
3. All the elements in an array have the same data type.
4. Struct fields are typically accessed using constant offsets.
5. Struct fields are typically not accessed in order, and are also not accessed using loops.
6. Struct fields are not typically all the same data type, or the same data width

Linked Lists and Binary Trees

Two common structures used when programming are linked lists and binary trees. These two structures in turn can be made more complicated in a number of ways. Shown in the images below are examples of a linked list structure and a binary tree structure.





Each node in a linked list or a binary tree contains some amount of data, and a pointer (or pointers) to other nodes. Consider the following asm code example:

```
loop_top:
cmp [ebp + 0], 10
je loop_end
mov ebp, [ebp + 4]
jmp loop_top
loop_end:
```

At each loop iteration, a data value at [ebp + 0] is compared with the value 10. If the two are equal, the loop is ended. If the two are not equal, however, the pointer in ebp is updated with a pointer at an offset from ebp, and the loop is continued. This is a classic linked-loop search technique. This is analagous to the following C code:

```
struct node
{
  int data,
  struct node * next
};

struct node x;
...
while(x.data != 10)
{
  x = x.next;
}
```

Binary trees are the same, except two different pointers will be used (the right and left branch pointers).

Objects and Classes

The Print Version page of the x86 Disassembly Wikibook is a stub. You can help by expanding this section.

Object-Oriented Programming

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 from C++. 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

Floating Point Numbers

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.

The x86 architecture does not have any registers specifically for floating point numbers, but it does have a special stack for them. The floating point stack is built directly into the processor, and has access speeds similar to those of ordinary registers. Notice that the FPU stack is not the same as the regular system stack.

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);
}
```

cl.exe doesn't 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 aren't 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 its 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 its own stack. We will see below that this is actually a correct assumption.

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 its 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 piece 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 definitely **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 its 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

Floating Point Examples

Example: Floating Point Arithmetic

Here is the C source code, and the GCC assembly listing of a simple C language function that performs simple floating-point arithmetic. Can you determine what the numerical values of LC5 and LC6 are?

```
__fastcall double MyFunction2(double x, double y, float z)
{
        return (x + 1.0) * (y + 2.0) * (z + 3.0);
}
```

```
.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
```

For this, we don't even need a floating-point number calculator, although you are free to use one if you wish (and if you can find a good one). LC5 is added to [ebp - 16], which we know to be y, and LC6 is added to [ebp - 24], which we know to be z. Therefore, LC5 is the number "2.0", and LC6 is the number "3.0". Notice that the **fld1** instruction automatically loads the top of the floating-point stack with the constant value "1.0".

Difficulties

Code Optimization

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.

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 won't even bother writing code to test x.

Control Flow Optimizations

Another set of optimization which can be performed either at the intermediate or at the code generation level are control flow optimizations. Most of these optimizations deal with the elimination of useless branches. Consider the following code:

```
if(A)
{
   if(B)
   {
      C;
   }
   else
   {
      D;
   }
   end_B:
}
else
{
   E;
}
end_A:
```

In this code, a simplistic compiler would generate a jump from the C block to end_B, and then another jump from end_B to end_A (to get around the E statements). Clearly jumping to a jump is inefficient, so optimizing compilers will generate a direct jump from block C to end_A.

This unfortunately will make the code more confused and will prevent a nice recovery of the original code. For complex functions, it's possible that one will have to consider the code made of only if()-goto; sequences, without being able to identify higher level statements like if-else or loops.

The process of identifying high level statement hierarchies is called "code structuring".

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 than 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 only drawback of a xor instruction is that it changes the processor flags, so it cannot be used between a comparison instruction and the corresponding conditional jump.

Loop Unwinding

When a loop needs to run for a small, but definite number of iterations, it is often better to **unwind the loop** in order to reduce the number of jump instructions performed, and in many cases prevent the processor's branch predictor from failing. Consider the following C loop, which calls the function `MyFunction()` 5 times:

```
for(x = 0; x < 5; x++)
{
  MyFunction();
}
```

Converting to assembly, we see that this becomes, roughly:

```
mov eax, 0
loop_top:
cmp eax, 5
jge loop_end
call _MyFunction
inc eax
jmp loop_top
```

Each loop iteration requires the following operations to be performed:

1. Compare the value in eax (the variable "x") to 5, and jump to the end if greater then or equal
2. Increment eax
3. Jump back to the top of the loop.

Notice that we remove all these instructions if we manually repeat our call to `MyFunction()`:

```
call _MyFunction
call _MyFunction
call _MyFunction
call _MyFunction
call _MyFunction
```

This new version not only takes up less disk space because it uses fewer instructions, but also runs faster because fewer instructions are executed. This process is called **Loop Unwinding**.

Inline Functions

The C and C++ languages allow the definition of an `inline` type of function. Inline functions are functions which are treated similarly to macros. During compilation, calls to an inline function are replaced with the body of that function, instead of performing a `call` instruction. In addition to using the `inline` keyword to declare an inline function, optimizing compilers may decide to make other functions inline as well.

Function inlining works similarly to loop unwinding for increasing code performance. A non-inline functionr requires a call instruction, several instructions to create a stack frame, and then several more instructions to destroy the stack frame and return from the function. By copying the body of the function instead of making a call, the size of the machine code increases, but the execution time *decreases*.

It is not necessarily possible to determine whether identical portions of code were created originally as macros, inline functions, or were simply copy and pasted. However, when disassembling it can make your work easier to separate these blocks out into separate inline functions, to help keep the code straight.

Optimization Examples

Example: Optimized vs Non-Optimized Code

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. Compare the listing file of this function when compiler optimizations are turned on and off.

```
/*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*/
 }
```

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.

Example: Manual Optimization

The following lines of assembly code are not optimized, but they can be optimized very easily. Can you find a way to optimize these lines?

```
mov   eax, 1
test   eax, eax
je     SHORT $L473
```

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)".

Example: Trace Variables

Here are the C code and the optimized assembly listing from the EuclidGCD function, from the example above. Can you determine which registers contain the variables **r** and **q**?

```
/*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*/
 }
```

```
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
```

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 (r). The instruction "mov ecx, edx" moves r into **ecx**, while q is not used for the rest of the loop, and is therefore discarded.

Example: Decompile Optimized Code

Below is the optimized listing file of the EuclidGCD function, presented in the examples above. Can you decompile this assembly code listing into equivalent "optimized" C code? How is the optimized version different in structure from the non-optimized version?

```
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
```

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.

Example: Instruction Pairings

Q

Why does the **dec**/**jne** combo operate faster than the equivalent **loopnz**?

A

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.

Example: Duff's Device

What does the following C code function do? Is it useful? Why or why not?

```
void MyFunction(int *arrayA, int *arrayB, cnt)
{
  switch(cnt % 6)
  {
    while(cnt ?= 0)
    {
      case 0:
        arrayA[cnt] = arrayB[cnt--];
      case 5:
        arrayA[cnt] = arrayB[cnt--];
      case 4:
        arrayA[cnt] = arrayB[cnt--];
      case 3:
        arrayA[cnt] = arrayB[cnt--];
      case 2:
        arrayA[cnt] = arrayB[cnt--];
      case 1:
        arrayA[cnt] = arrayB[cnt--];
    }
  }
}
```

This piece of code is known as a **Duff's device** or "Duff's machine". It is used to partially unwind a loop for efficiency. Notice the strange way that the `while()` is nested inside the `switch` statement? Two arrays of integers are passed to the function, and at each iteration of the while loop, 6 consecutive elements are copied from `arrayB` to `arrayA`. The switch statement, since it is outside the while loop, only occurs at the beginning of the function. The modulo is taken of the variable `cnt` with respect to 6. If cnt is not evenly divisible by 6, then the modulo statement is going to start the loop off somewhere in the middle of the rotation, thus preventing the loop from causing a buffer overflow without having to test the current count after each iteration.

Duff's Device is considered one of the more efficient general-purpose methods for copying strings, arrays, or data streams.

Code Obfuscation

Code Obfuscation

Code Obfuscation is the act of making the assembly code or machine code of a program more difficult to disassemble or decompile. The term "obfuscation" is typically used to suggest a deliberate attempt to add difficulty, but many other practices will cause code to be obfuscated without that being the intention. Software vendors may attempt to obfuscate or even encrypt code to prevent reverse engineering efforts. There are many different types of obfuscations. Notice that many code optimizations (discussed in the previous chapter) have the side-effect of making code more difficult to read, and therefore optimizations act as obfuscations.

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 put in a hard-to read order.
- Code instructions which are used in a non-obvious way.

This chapter will try to examine some common methods of obfuscating code, but will not necessarily delve into methods to break the obfuscation.

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.

x86 NetBurst Architecture

The Intel **NetBurst Architecture** divides an x86 processor into 2 distinct parts: the supporting hardware, and the primitive core processor. The primitive 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 primitive 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 executed in a single instruction period.

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.

Non-Intuitive Instructions

Optimizing compilers frequently will use instructions that are not intuitive. Some instructions can perform tasks for which they were not designed, typically as a helpful side effect. Sometimes, one instruction can perform a task more quickly then other specialized instructions can.

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.

Here are some examples. The code in the left column operates more quickly then the code on the right, but performs exactly the same tasks.

```
xor eax, eax
```

```
mov eax, 0
```

```
shl eax, 3
```

```
mul eax, 8
```

Sometimes such transformations could be made to make the analysis more difficult:

```
push $next_instr
jmp $some_function
$next_instr:...
```

```
call $some_function
```

```
pop eax
jmp eax
```

```
retn
```

Common 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 multiplication 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 **mul** 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".

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. One important fact is that decimal digits are not present in the resulting number. For example:

```
mov al, 31 ; 00011111
 shr al, 1  ; 00001111 = 15, not 15.5
```

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.

As an example, consider this code:

```
mov ebx, eax
mov eax, 0
```

Here, the value in `eax` is stored in `ebx`, and then `eax` is loaded with the value zero. We can perform the same operation, but using `xchg` and `xor` instead:

```
xchg eax, ebx
xor eax, eax
```

It may surprise you to learn that the second code example operates significantly faster then the first one does.

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 are a way of reordering code so that it performs exactly the same task but becomes more difficult to trace and disassemble. 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

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 doesn't do anything and looks useless. A program filled with opaque predicates will take more time to decipher, because the disassembler will take more time reading through useless, distraction code.

Code Encryption

Code can be encrypted, just like any other type of data, except that code can also work to encrypt and decrypt *itself.* Encrypted programs cannot be directly disassembled. However, such a program can also not be run directly because the encrypted opcodes cannot be interpreted properly by the CPU. For this reason, an encrypted program must contain some sort of method for decrypting itself prior to operation.

The most basic method is to include a small stub program that decrypts the remainder of the executable, and then passes control to the decrypted routines.

Disassembling Encrypted Code

To disassemble an encrypted executable, you must first determine how the code is being decrypted. Code can be decrypted in one of two primary ways:

1. All at once. The entire code portion is decrypted in a single pass, and left decrypted during execution. Using a debugger, allow the decryption routine to run completely, and then dump the decrypted code into a file for further analysis.
2. By Block. The code is encrypted in separate blocks, where each block may have a separate encryption

key. Blocks may be decrypted before use, and re-encrypted again after use. Using a debugger, you can attempt to capture all the decryption keys and then use those keys to decrypt the entire program at once later, or you can wait for the blocks to be decrypted, and then dump the blocks individually to a separate file for analysis.

Debugger Detectors

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.

This subject is peripheral to the narrative of this book, and the section should be considered an optional one for most readers.

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. For OllyDbg, there are many plugins available which hide the debugger from this and many other APIs.

PEB Debugger Check

The Process Environment Block stores the value that IsDebuggerPresent queries to determine its return value. To avoid suspicion, some programmers access the value directly from the PEB instead of calling the API function. The following code snippet shows how to access the value:

```
mov eax, fs:[30h]
mov eax, byte [eax+2]
test eax, eax
jne @DebuggerDetected
```

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.

Notice that on preemptive multithreading systems, such as modern Windows or Linux systems will switch away from your program to run other programs. This is called thread switching. If the system has many threads to run, or if some threads are hogging processor time, your program may detect a long delay and may falsely determine that the program is being debugged.

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(). [3] (http://www.securityfocus.com/bid/10742) 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.

Decompiler Theory

Disassembler Theory

Decompiler Theory

Resources

Resources

Wikimedia Resources

Wikibooks

- X86 Assembly
- Subject:Assembly Language
- Compiler Construction
- Floating Point
- C Programming
- C++ Programming

Wikipedia

External Resources

External Links

- The MASM Project: http://www.masm32.com/
- Randall Hyde's Homepage: http://www.cs.ucr.edu/~rhyde/
- Borland Turbo Assembler: http://info.borland.com/borlandcpp/cppcomp/tasmfact.html
- NASM Project Homepage: http://nasm.sourceforge.net/wakka.php?wakka=HomePage
- FASM Homepage: http://flatassembler.net/
- DCC Decompiler: [4] (http://www.itee.uq.edu.au/~cristina/dcc.html)
- Boomerang Decompiler Project: [5] (http://boomerang.sourceforge.net/)
- Microsoft debugging tools main page:

  http://www.microsoft.com/whdc/devtools/debugging/default.mspx

- Solaris observation and debugging tools main page:

  http://www.opensolaris.org/os/community/dtrace/
  http://www.opensolaris.org/os/community/mdb/

- Free Debugging Tools, Static Source Code Analysis Tools, Bug Trackers (http://www.thefreecountry.com/programming/debuggers.shtml)
- Microsoft Developers Network (MSDN): http://msdn.microsoft.com
- Gareth Williams: http: //garethhegeek.ga.funpic.de/
- B. Luevelsmeyer "PE Format Description":http://www.cs.bilkent.edu.tr/~hozgur/PE.TXT PE format description
- MSDN Calling Convention page: [6] (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/html/_core_calling_conventions_topics.asp)
- Dictionary of Algorithms and Data Structures (http://www.nist.gov/dads/)
- Charles Petzold's Homepage: http://www.charlespetzold.com/
- Donald Knuth's Homepage: http://www-cs-faculty.stanford.edu/~knuth/
- "THE ISA AND PC/104 BUS" (http://www.techfest.com/hardware/bus/isa_sokos.htm) by Mark Sokos 2000
- "Practically Reversing CRC" (http://blog.w-nz.com/archives/2005/07/) by Bas Westerbaan 2005

- "CRC and how to Reverse it" (http://www.woodmann.com/fravia/crctut1.htm) by anarchriz 1999
- *"Reverse Engineering is a Way of Life" (http://speakeasy.org/~russotto/)* by Matthew Russotto
- *"the Reverse and Reengineering Wiki" (http://www.program-transformation.org/Transform /ReengineeringWiki)*

Books

- Eilam, Eldad. "Reversing: Secrets of Reverse Engineering." 2005. Wiley Publishing Inc. ISBN 0764574817
- Hyde, Randall. "The Art of Assembly Language," No Starch, 2003 ISBN 1886411972
- Aho, Alfred V. et al. "Compilers: Principles, Techniques and Tools," Addison Wesley, 1986. ISBN: 0321428900
- Steven Muchnick, "Advanced Compiler Design & Implementation," Morgan Kaufmann Publishers, 1997. ISBN 1-55860-320-4
- Kernighan and Ritchie, "The C Programming Language", 2nd Edition, 1988, Prentice Hall.
- Petzold, Charles. "Programming Windows, Fifth Edition," Microsoft Press, 1999
- Hart, Johnson M. "Win32 System Programming, Second Edition," Addison Wesley, 2001
- Gordon, Alan. "COM and COM+ Programming Primer," Prentice Hall, 2000
- Nebbett, Gary. "Windows NT/2000 Native API Reference," Macmillan, 2000
- Levine, John R. "Linkers and Loaders," Morgan-Kauffman, 2000
- Knuth, Donald E. "The Art of Computer Programming," Vol 1, 1997, Addison Wesley.
- *MALWARE: Fighting Malicious Code,* by Ed Skoudis; Prentice Hall, 2004
- *Maximum Linux Security, Second Edition,* by Anonymous; Sams, 2001

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject

matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding

the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved

by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document

is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.