The Mac® Hacker's Handbook

Charlie Miller
Dino Dai Zovi

The Mac® Hacker's Handbook

Charlie Miller
Dino A. Dai Zovi

Wiley Publishing, Inc.

The Mac® Hacker's Handbook

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published simultaneously in Canada

ISBN: 978-0-470-39536-3

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging-in-Publication Data is available from the publisher.

I'd like to dedicate this book to the security research community and everyone who is passionate about advancing the state of offensive and defensive security knowledge.

— Dino A. Dai Zovi

About the Authors

Charlie Miller is Principal Analyst at Independent Security Evaluators. He was the first person to publically create a remote exploit against Apple's iPhone and the G1 Google phone running Android. He has discovered flaws in numerous applications on various operating systems. He was the winner of the 2008 PwnToOwn contest for breaking into a fully patched MacBook Air. He has spoken at numerous information-security conferences and is author of *Fuzzing for Software Security Testing and Quality Assurance* (Artech House, 2008). He was listed as one of the top 10 hackers of 2008 by *Popular Mechanics* magazine, and has a PhD from the University of Notre Dame.

Dino Dai Zovi is Chief Scientist at a private information security firm. Mr. Dai Zovi is perhaps best known in the security and Mac communities for winning the first Pwn2Own contest at CanSecWest 2007 by discovering and exploit- ing a new vulnerability in Apple's QuickTime in one night to compromise a fully patched MacBook Pro. He previously specialized in software penetration testing in roles at Matasano Security, @stake, and Sandia National Laboratories. He is an invited speaker at information security conferences around the world, a coauthor of *The Art of Software Security Testing: Identifying Software Security Flaws* (Addison-Wesley, 2006) and was named one of the 15 Most Influential People in Security by *eWEEK* in 2007.

Credits

Executive Editor
Carol Long

Development Editor
Christopher J. Rivera

Technical Editor
Ron Krutz

Production Editor
Elizabeth Ginns Britten

Copy Editor
Candace English

Editorial Manager
Mary Beth Wakefield

Production Manager
Tim Tate

Vice President and Executive Group Publisher
Richard Swadley

Vice President and Executive Publisher
Barry Pruett

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Lynsey Stanford

Compositor
Jeffrey Lytle,
Happenstance Type-O-Rama

Proofreader
Justin Neely, Word One

Indexer
Jack Lewis

Cover Illustration
Michael E. Trent

Cover Designer
Michael E. Trent

Acknowledgments

I'd like to thank my wife Andrea for not getting too upset when I locked myself away at night to work on the book after the kids went to bed. I'd also like to thank my two sons, Theo and Levi, for being good kids and keeping a smile on my face. Finally, I'd like to thank ISE for giving me time to do research for the book, and the following people for donating their time to look at early drafts of it: Dave Aitel, Thomas Ptacek, Thomas Dullien, and Nate McFeters.

— Charlie Miller

I'd like to thank my friends for their support and patience while I was working on this book and lacking a normal social life for the warmer half of the year. I'd also like to thank the members of the Apple Product Security team for their diligence in addressing the security issues that I have reported to them over the years, as well as Apple for creating an operating system and computers that are a joy to use. Finally, I'd like to thank our volunteer reviewers, Dave Aitel, Halvar Flake, and Thomas Ptacek, for their advice and comments.

— Dino A. Dai Zovi

Contents

Foreword

For better or worse, there are moments in our lives that we can visualize with startling clarity. Sometimes momentous and other times trivial, we're able to completely recall these snippets of our past even if we can't remember the day or context. In my life, there's one moment I'd like to call trivial, but the truth is, it was likely more central in establishing my eventual technology career than I care to admit at social gatherings.

I think it was the early 1980s, but that's mostly irrelevant. My best friend's parents recently purchased an Apple II (plus, I think), making my friend the first person I knew with a computer in his house. One day we noticed a seam on the top of the plastic case; we slid the bulking green screen monitor to the side and removed the panel on the top. For the first time, we peered into the inner guts of an actual working computer. This was definitely before the release of WarGames, likely before I'd ever heard of hacking, and long before "hacker" became synonymous with "criminal" in the mass media. We lifted that plastic lid and stared at the copper and black components on the field of green circuit boards before us. We were afraid to touch anything, but for the first time, the walls between hardware and software shattered for our young minds, opening up a new world of possibilities. This was something we could *touch*, manipulate, and, yes, break.

My young computer career began with those early Apples (and Commodores). We spent countless hours exploring their inner workings; from BASIC to binary math, and more than our fair share of games (for the record, the Apple joystick was terrible). Early on I realized I enjoyed breaking things just as much, if not more than, creating them. By feeling around the seams of software and systems, learning where they bent, cracked, and failed, I could understand them in ways just not possible by coloring between the lines.

The very first Mac I could buy was an early Mac Mini I purchased mostly for research purposes. I quickly realized that Mac OS X was a hacker's delight of an operating system. Beautiful and clean compared to my many years on Windows, with a Unix terminal a click away. Here was a box I could run Microsoft Office on that came with Apache by default and still held full man pages. As I delved into Applescript, plists, DMGs, and the other minutia of OS X, I was amazed by the capabilities of the operating system, and the breadth and depth of tools available.

But as I continued to switch completely over to Apple, especially after the release of Intel Macs, my fingers started creeping around for those cracks at the edges again. I wasn't really worried about viruses, but, as a security professional, I started wondering if this was by luck or design. I read the Apple documentation and realized fairly early that there wasn't a lot of good information on how OS X worked from a security standpoint, other than some configuration guides and marketing material.

Mac security attitudes have changed a fair bit since I purchased that first Mac Mini. As Macs increase in popularity, they face more scrutiny. Windows switchers come with questions and habits, more security researchers use Macs in their day-to-day work, the press is always looking to knock Apple down a notch, and the bad guys won't fail to pounce on any profitable opportunity. But despite this growing attention, there are few resources for those who want to educate themselves and better understand the inner workings of the operating system on which they rely.

That's why I was so excited when Dino first mentioned he and Charlie were working on this book. Ripping into the inner guts of Mac OS X and finding those edges to tear apart are the only ways to advance the security of the platform. Regular programming books and system overviews just don't look at any operating system from the right perspective; we need to know how something breaks in order to make it stronger. And, as any child (or hacker) will tell you, breaking something is the most exhilarating way to learn.

If you are a security professional, this book is one of the best ways to understand the strengths and weaknesses of Mac OS X. If you are a programmer, this book will not only help you write more secure code, but it will also help you in your general coding practices. If you are just a Mac enthusiast, you'll learn how hackers look at our operating system of choice and gain a better understanding of its inner workings. Hopefully Apple developers will use this to help harden the operating system; making the book obsolete with every version. Yes, maybe a few bad guys will use it to write a few exploits, but the benefits of having this knowledge far outweigh the risks.

For us hackers, even those of us of limited skills, this book provides us with a roadmap for exploring those edges, finding those cracks, and discovering new possibilities. For me, it's the literary equivalent of sliding that beige plastic cover off my childhood friend's first Apple and gazing at the inner workings.

—Rich Mogull
Security Editor at TidBITS and Analyst at Securosis

Introduction

As Mac OS X continues to be adopted by more and more users, it is important to consider the security (or insecurity) of the devices running it. From a security perspective, Apple has led a relatively charmed existence so far. Mac OS X computers have not had any significant virus or worm outbreaks, making them a relatively safe computing platform. Because of this, they are perceived by most individuals to be significantly more secure than competing desktop operating systems, such as Windows XP or Vista.

Overview of the Book and Technology

Is this perception of security justified, or has Mac OS X simply benefited from its low profile up to this point? This book offers you a chance to answer this question for yourself. It provides the tools and techniques necessary to analyze thoroughly the security of computers running the Mac OS X operating system. It details exactly what Apple has done right in the design and implementation of its code, as well as points out deficiencies and weaknesses. It teaches how attackers look at Mac OS X technologies, probe for weaknesses, and succeed in compromising the system. This book is not intended as a blueprint for malicious attackers, but rather as an instrument so the good guys can learn what the bad guys already know. Penetration testers and other security analysts can and should use this information to identify risks and secure the Macs in their environments.

Keeping security flaws secret does not help anybody. It is important to understand these flaws and point them out so future versions of Mac OS X will be more secure. It is also vital to understand the security strengths and weaknesses of the operating system if we are to defend properly against attack, both now and in the future. Information is power, and this book empowers its readers by providing the most up-to-date and cutting-edge Mac OS X security research.

How This Book Is Organized

This book is divided into four parts, roughly aligned with the steps an attacker would have to take to compromise a computer: Background, Vulnerabilities, Exploitation, and Post-Exploitation. The first part, consisting of Chapters 1–3, contains introductory material concerning Mac OS X. It points out what makes this operating system different from Linux or Windows and demonstrates the tools that will be needed for the rest of the book. The next part, consisting of Chapters 4–6, demonstrates the tools and techniques necessary to identify security vulnerabilities in the operating system and applications running on it. Chapters 7–10 make up the next part of the book. These chapters illustrate how attackers can take the weaknesses found in the earlier chapters and turn them into functional exploits, giving them the ability to compromise vulnerable machines. Chapters 11 and 12 make up the last part of the book, which deals with what attackers may do after they have exploited a machine and techniques they can use to maintain continued access to the compromised machines.

Chapter 1 begins the book with the basics of the way Mac OS X is designed. It discusses how it originated from BSD and the changes that have been made in it since that time. Chapter 1 gives a brief introduction to many of the tools that will be needed in the rest of the book. It highlights the differences between Mac OS X and other operating systems and takes care to demonstrate how to perform common tasks that differ among the operating systems. Finally, it outlines and analyzes some of the security improvements made in the release of Leopard, the current version of Mac OS X.

Chapter 2 covers some uncommon protocols and file formats used by Mac OS X. This includes a description of how Bonjour works, as well as an inside look at the Mac OS X implementation, mDNSResponder. It also dissects the QuickTime file format and the RTSP protocol utilized by QuickTime Player.

Chapter 3 examines what portions of the operating system process attacker-supplied data, known as the attack surface. It begins by looking in some detail at what services are running by default on a typical Mac OS X computer and examines the difficulties in attacking these default services. It moves on to consider the client-side attack surface, all the code that can be executed if an attacker can get a client program such as Safari to visit a server the attacker controls, such as a malicious website.

Chapter 4 dives into the world of debugging in a Mac OS X environment. It shows how to follow along to see what applications are doing internally. It covers in some detail the powerful DTrace mechanism that was introduced in Leopard. It also outlines the steps necessary to capture code-coverage information using the Pai Mei reverse-engineering framework.

Chapter 5 demonstrates how to find security weaknesses in Mac OS X software. It talks about how you can look for bugs in the source code Apple makes available or use a black-box technique such as fuzzing. It includes detailed instructions for performing either of these methods. Finally, it shows some tricks to take advantage of the way Apple develops its software, which can help find bugs it doesn't know about or give early warning of those it does.

Chapter 6 discusses reverse engineering in Mac OS X. Given that most of the code in Mac OS X is available in binary form only, this chapter discusses how this software works statically. It also highlights some differences that arise in reverse engineering code written in Objective-C, which is quite common in Mac OS X binaries but rarely seen otherwise.

Chapter 7 begins the exploitation part of the book. It introduces the simplest of buffer-overflow attacks, the stack overflow. It outlines how the stack is laid out for both PowerPC and x86 architectures and how, by overflowing a stack buffer, an attacker can obtain control of the vulnerable process.

Chapter 8 addresses the heap overflow, the other common type of exploit. This entails describing the way the Mac OS X heap and memory allocations function. It shows techniques where overwriting heap metadata allows an attacker to gain complete control of the application. It finishes by showing how to arrange the heap to overwrite other important application data to compromise the application.

Chapter 9 addresses exploit payloads. Now that you know how to get control of the process, what can you do? It demonstrates a number of different possible shellcodes and payloads for both PowerPC and x86 architectures, ranging from simple to advanced.

Chapter 10 covers real-world exploitation, demonstrating a large number of advanced exploitation topics, including many in-depth example exploits for Tiger and Leopard on both PowerPC and x86. If Chapters 7–9 were the theory of attack, then this chapter is the practical aspect of attack.

Chapter 11 covers how to inject code into running processes using Mac OS X–specific hooking techniques. It provides all the code necessary to write and test such payloads. It also includes some interesting code examples of what an attacker can do, including spying on iChat sessions and reading encrypted network traffic.

Chapter 12 addresses the topic of rootkits, or code an attacker uses to hide their presence on a compromised system. It illustrates how to write basic kernel-level drivers and moves on to examples that will hide files from unsuspecting users at the kernel level. It finishes with a discussion of Mac OS X–specific rootkit techniques, including hidden in-kernel Mach RPC servers, network kernel extensions for remote access, and VT-x hardware virtual-machine hypervisor rootkits for advanced stealth.

Who Should Read This Book

This book is written for a wide variety of readers, ranging from Mac enthusiasts to hard-core security researchers. Those readers already knowledgeable about Mac OS X but wanting to learn more about the security of the system may want to skip to Chapter 4. Conversely, security researchers may find the first few chapters the most useful, as those chapters reveal how to use the OS X–related skills they already possess.

While the book may be easier to comprehend if you have some experience writing code or administering Mac OS X computers, no experience is necessary. It starts from the very basics and slowly works up to the more-advanced topics. The book is careful to illustrate the points it is making with many examples, and outlines exactly how to perform the steps required. The book is unique in that, although anybody with enthusiasm for the subject can pick it up and begin reading it, by the end of the book the reader will have a world-class knowledge of the security of the Mac OS X operating system.

Tools You Will Need

For the most part, all you need to follow along with this book is a computer with Mac OS X Leopard installed. Although many of the techniques and examples will work in earlier versions of Mac OS X, they are designed for Leopard.
To perform the techniques illustrated in Chapter 6, a recent version of IDA Pro is required. This is a commercial tool that must be run in Windows and can be purchased at `http://www.hex-rays.com`. The remaining tools either come on supplemental disks, such as Xcode does, or are freely available online or at this book's website.

What's on the Website

This book includes a number of code samples. The small and moderately sized examples are included directly in this book. But to save you from having to type these in yourself, all the code samples are also available for download at `www.wiley.com/go/machackershandbook`. Additionally, some long code samples that are omitted from the book are available on the site, as are any other tools developed for the book.

Final Note

We invite you to dive right in and begin reading. We think there is something in this book for just about everyone who loves Mac OS X. I know we learned a lot in researching and writing this book. If you have comments, questions, hate mail, or anything else, please drop us a line and we'd be happy to discuss our favorite operating system with you.

Part

I

Mac OS X Basics

CHAPTER

1

Mac OS X Architecture

This chapter begins by addressing many of the basics of a Mac OS X system. This includes the general architecture and the tools necessary to deal with the architecture. It then addresses some of the security improvements that come with version 10.5 “Leopard”, the most recent version of Mac OS X. Many of these security topics will be discussed in great detail throughout this book.

Basics

Before we dive into the tools, techniques, and security of Mac OS X, we need to start by discussing how it is put together. To understand the details of Leopard, you need first to understand how it is built, from the ground up. As depicted in Figure 1-1, Mac OS X is built as a series of layers, including the XNU kernel and the Darwin operating system at the bottom, and the Aqua interface and graphical applications on the top. The important components will be discussed in the following sections.

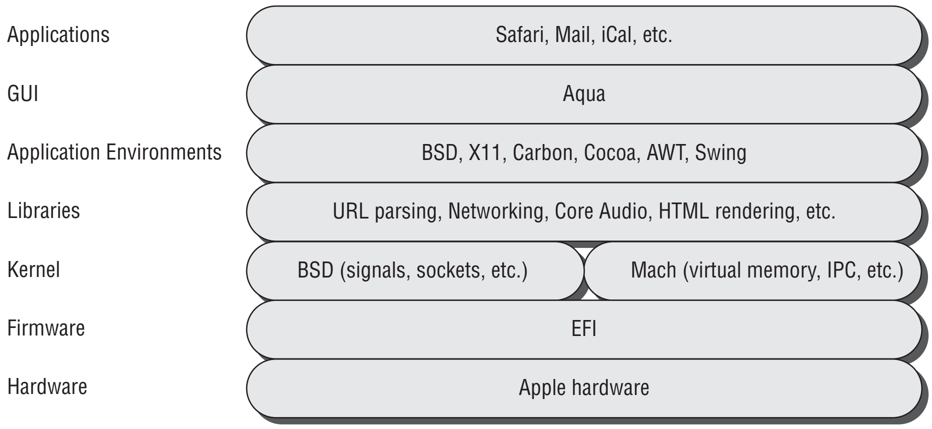
Applications
Safari, Mail, iCal, etc.
GUI
Aqua
Application Environments
BSD, X11, Carbon, Cocoa, AWT, Swing
Libraries
URL parsing, Networking, Core Audio, HTML rendering, etc.
Kernel
BSD (signals, sockets, etc.)
Mach (virtual memory, IPC, etc.)
Firmware
EFI
Hardware
Apple hardware

Figure 1-1: Basic architecture of a Mac OS X system

XNU

The heart of Mac OS X is the XNU kernel. XNU is basically composed of a Mach core (covered in the next section) with supplementary features provided by Berkeley Software Distribution (BSD). Additionally, XNU is responsible for providing an environment for kernel drivers called the I/O Kit. We'll talk about each of these in more detail in upcoming sections. XNU is a Darwin package, so all of the source code is freely available. Therefore, it is completely possible to install the same kernel used by Mac OS X on any machine with supported hardware; however, as Figure 1-1 illustrates, there is much more to the user experience than just the kernel.

From a security researcher's perspective, Mac OS X feels just like a FreeBSD box with a pretty windowing system and a large number of custom applications. For the most part, applications written for BSD will compile and run without modification on Mac OS X. All the tools you are accustomed to using in BSD are available in Mac OS X. Nevertheless, the fact that the XNU kernel contains all the Mach code means that some day, when you have to dig deeper, you'll find many differences that may cause you problems and some you may be able to leverage for your own purposes. We'll discuss some of these important differences briefly; for more detailed coverage of these topics, see *Mac OS X Internals: A Systems Approach* (Addison-Wesley, 2006).

Mach

Mach, developed at Carnegie Mellon University by Rick Rashid and Avie Tevanian, originated as a UNIX-compatible operating system back in 1984. One of its primary design goals was to be a microkernel; that is, to minimize the amount of code running in the kernel and allow many typical kernel functions, such as file system, networking, and I/O, to run as user-level Mach tasks. In earlier Mach-based UNIX systems, the UNIX layer ran as a server in a separate task. However, in Mac OS X, Mach and the BSD code run in the same address space.

In XNU, Mach is responsible for many of the low-level operations you expect from a kernel, such as processor scheduling and multitasking and virtual-memory management.

BSD

The kernel also involves a large chunk of code derived from the FreeBSD code base. As mentioned earlier, this code runs as part of the kernel along with Mach and uses the same address space. The FreeBSD code within XNU may differ significantly from the original FreeBSD code, as changes had to be made for it to coexist with Mach. FreeBSD provides many of the remaining operations the kernel needs, including

- Processes
- Signals
- Basic security, such as users and groups
- System call infrastructure
- TCP/IP stack and sockets
- Firewall and packet filtering

To get an idea of just how complicated the interaction between these two sets of code can be, consider the idea of the fundamental executing unit. In BSD the fundamental unit is the process. In Mach it is a Mach thread. The disparity is settled by each BSD-style process being associated with a Mach task consisting of exactly one Mach thread. When the BSD fork() system call is made, the BSD code in the kernel uses Mach calls to create a task and thread structure. Also, it is important to note that both the Mach and BSD layers have different security models. The Mach security model is based on port rights, and the BSD model is based on process ownership. Disparities between these two models have resulted in a number of local privilege-escalation vulnerabilities. Additionally, besides typical system cells, there are Mach traps that allow user-space programs to communicate with the kernel.

I/O Kit

I/O Kit is the open-source, object-oriented, device-driver framework in the XNU kernel and is responsible for the addition and management of dynamically loaded device drivers. These drivers allow for modular code to be added to the kernel dynamically for use with different hardware, for example. The available drivers are usually stored in the /System/Library/Extensions/ directory or a subdirectory. The command kextstat will list all the currently loaded drivers,

```
$ kextstat
Index Refs Address    Size       Wired      Name (Version) <Linked
Against>
    1    1 0x0        0x0        0x0        com.apple.kernel (9.3.0)
    2   55 0x0        0x0        0x0        com.apple.kpi.bsd (9.3.0)
    3    3 0x0        0x0        0x0        com.apple.kpi.dsep (9.3.0)
    4   74 0x0        0x0        0x0        com.apple.kpi.iokit (9.3.0)
    5   79 0x0        0x0        0x0        com.apple.kpi.libkern
(9.3.0)
    6   72 0x0        0x0        0x0        com.apple.kpi.mach (9.3.0)
    7   39 0x0        0x0        0x0        com.apple.kpi.unsupported
(9.3.0)
    8    1 0x0        0x0        0x0
com.apple.iokit.IONVRAMFamily (9.3.0)
    9    1 0x0        0x0        0x0        com.apple.driver.AppleNMI
(9.3.0)
   10    1 0x0        0x0        0x0
com.apple.iokit.IOSystemManagementFamily (9.3.0)
   11    1 0x0        0x0        0x0
com.apple.iokit.ApplePlatformFamily (9.3.0)
   12   31 0x0        0x0        0x0        com.apple.kernel.6.0 (7.9.9)
   13    1 0x0        0x0        0x0        com.apple.kernel.bsd (7.9.9)
   14    1 0x0        0x0        0x0        com.apple.kernel.iokit
(7.9.9)
   15    1 0x0        0x0        0x0        com.apple.kernel.libkern
(7.9.9)
   16    1 0x0        0x0        0x0        com.apple.kernel.mach
(7.9.9)
   17   17 0x2e2bc000 0x10000    0xf000     com.apple.iokit.IOPCIFamily
(2.4.1) <7 6 5 4>
   18   10 0x2e2d2000 0x4000     0x3000     com.apple.iokit.IOACPIFamily
(1.2.0) <12>
   19    3 0x2e321000 0x3d000    0x3c000
com.apple.driver.AppleACPIPlatform (1.2.1) <18 17 12 7 5 4>
…
```

Many of the entries in this list say they are loaded at address zero. This just means they are part of the kernel proper and aren't really device drivers—i.e., they cannot be unloaded. The first actual driver is number 17.

Besides kextstat, there are other functions you'll need to know for loading and unloading these drivers. Suppose you wanted to find and load the driver associated with the MS-DOS file system. First you can use the kextfind tool to find the correct driver.

```
$ kextfind -bundle-id -substring 'msdos'
/System/Library/Extensions/msdosfs.kext
```

Now that you know the name of the kext bundle to load, you can load it into the running kernel.

```
$ sudo kextload /System/Library/Extensions/msdosfs.kext
kextload: /System/Library/Extensions/msdosfs.kext loaded successfully
```

It seemed to load properly. You can verify this and see where it was loaded.

```
$ kextstat | grep msdos
  126    0 0x346d5000 0xc000     0xb000
com.apple.filesystems.msdosfs (1.5.2) <7 6 5 2>
```

It is the 126th driver currently loaded. There are zero references to it (not surprising, since it wasn't loaded before we loaded it). It has been loaded at address 0x346d5000 and has size 0xc000. This driver occupies 0xb000 wired bytes of kernel memory. Next it lists the driver's name and version. It also lists the index of other kernel extensions that this driver refers to—in this case, looking at the full listing of kextstat, we see it refers to the "unsupported" mach, libkern, and bsd drivers. Finally, we can unload the driver.

```
$ sudo kextunload com.apple.filesystems.msdosfs
kextunload: unload kext /System/Library/Extensions/msdosfs.kext
succeeded
```

Darwin and Friends

A kernel without applications isn't very useful. That is where Darwin comes in. Darwin is the non-Aqua, open-source core of Mac OS X. Basically it is all the parts of Mac OS X for which the source code is available. The code is made available in the form of a package that is easy to install. There are hundreds of available Darwin packages, such as X11, GCC, and other GNU tools. Darwin provides many of the applications you may already use in BSD or Linux for Mac OS X. Apple has spent significant time integrating these packages into their operating system so that everything behaves nicely and has a consistent look and feel when possible.

On the other hand, many familiar pieces of Mac OS X are not open source. The main missing piece to someone running just the Darwin code will be Aqua, the Mac OS X windowing and graphical-interface environment. Additionally, most of the common high-level applications, such as Safari, Mail, QuickTime, iChat, etc., are not open source (although some of their components *are* open source). Interestingly, these closed-source applications often rely on open-source software, for example, Safari relies on the WebKit project for HTML and JavaScript rendering. For perhaps this reason, you also typically have many more symbols in these applications when debugging than you would in a Windows environment.

Tools of the Trade

Many of the standard Linux/BSD tools work on Mac OS X, but not all of them. If you haven't already, it is important to install the Xcode package, which contains the system compiler (gcc) as well as many other tools, like the GNU debugger gdb. One of the most powerful tools that comes on Mac OS X is the object file displaying tool (otool). This tool fills the role of ldd, nm, objdump, and similar tools from Linux. For example, using otool you can use the `-L` option to get a list of the dynamically linked libraries needed by a binary.

```
$ otool -L /bin/ls
/bin/ls:
/usr/lib/libncurses.5.4.dylib (compatibility version 5.4.0, current
version 5.4.0)
/usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version
1.0.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
111.0.0)
```

To get a disassembly listing, you can use the `-tv` option.

```
$ otool -tv /bin/ps
/bin/ps:
(__TEXT,__text) section
00001bd0        pushl   $0x00
00001bd2        movl    %esp,%ebp
00001bd4        andl    $0xf0,%esp
00001bd7        subl    $0x10,%esp
…
```

You'll see many references to other uses for otool throughout this book.

Ktrace/DTrace

You must be able to trace execution flow for processes. Before Leopard, this was the job of the ktrace command-line application. ktrace allows kernel trace logging for the specified process or command. For example, tracing the system calls of the ls command can be accomplished with

```
$ ktrace -tc ls
```

This will create a file called ktrace.out. To read this file, run the kdump command.

```
$ kdump
   918 ktrace   RET   ktrace 0
   918 ktrace   CALL  execve(0xbffff73c,0xbffffd14,0xbffffd1c)
   918 ls       RET   execve 0
   918 ls       CALL  issetugid
   918 ls       RET   issetugid 0
   918 ls       CALL
__sysctl(0xbffff7cc,0x2,0xbffff7d4,0xbffff7c8,0x8fe45a90,0xa)
   918 ls       RET   __sysctl 0
   918 ls       CALL  __sysctl(0xbffff7d4,0x2,0x8fe599bc,0xbffff878,0,0)
   918 ls       RET   __sysctl 0
   918 ls       CALL
__sysctl(0xbffff7cc,0x2,0xbffff7d4,0xbffff7c8,0x8fe45abc,0xd)
   918 ls       RET   __sysctl 0
   918 ls       CALL  __sysctl(0xbffff7d4,0x2,0x8fe599b8,0xbffff878,0,0)
   918 ls       RET   __sysctl 0
…
```

For more information, see the man page for ktrace.

In Leopard, ktrace is replaced by DTrace. DTrace is a kernel-level tracing mechanism. Throughout the kernel (and in some frameworks and applications) are special DTrace probes that can be activated. Instead of being an application with some command-line arguments, DTrace has an entire language, called D, to control its actions. DTrace is covered in detail in Chapter 4, "Tracing and Debugging," but we present a quick example here as an appetizer.

```
$ sudo dtrace -n 'syscall:::entry {@[execname] = count()}'
dtrace: description 'syscall:::entry ' matched 427 probes
^C

  fseventsd                                                         3
  socketfilterfw                                                    3
  mysqld                                                            6
  httpd                                                             8
  pvsnatd                                                           8
  configd                                                          11
  DirectoryServic                                                  14
  Terminal                                                         17
  ntpd                                                             21
  WindowServer                                                     27
  mds                                                              33
  dtrace                                                           38
  llipd                                                            60
  SystemUIServer                                                   69
  launchd                                                         182
  nmblookup                                                       288
  smbclient                                                       386
  Finder                                                         5232
  Mail                                                           5352
```

Here, this one line of D within the DTrace command keeps track of the number of system calls made by processes until the user hits Ctrl+C. The entire functionality of ktrace can be replicated with DTrace in just a few lines of D. Being able to peer inside processes can be very useful when bug hunting or reverse-engineering, but there will be more on those topics later in the book.

Objective-C

Objective-C is the programming language and runtime for the Cocoa API used extensively by most applications within Mac OS X. It is a superset of the C programming language, meaning that any C program will compile with an Objective-C compiler. The use of Objective-C has implications when applications are being reverse-engineered and exploited. More time will be spent on these topics in the corresponding chapters.

One of the most distinctive features of Objective-C is the way object-oriented programming is handled. Unlike in standard C++, in Objective-C, class methods are not called directly. Rather, they are sent a message. This architecture allows for *dynamic binding*; i.e., the selection of method implementation occurs at runtime, not at compile time. When a message is sent, a runtime function looks at the receiver and the method name in the message. It identifies the receiver's implementation of the method by the name and executes that method.

The following small example shows the syntactic differences between C++ and Objective-C from a source-code perspective.

```
#include <objc/Object.h>
@interface Integer : Object
{
   int integer;
}

- (int) integer;
- (id) integer: (int) _integer;
@end
```

Here an interface is defined for the class Integer. An interface serves the role of a declaration. The hyphen character indicates the class's methods.

```
#import "Integer.h"
@implementation Integer
- (int) integer
{
   return integer;
}

- (id) integer: (int) _integer
```

```
 {
    integer = _integer;
 }
 @end
```

Objective-C source files typically use the .m file extension. Within Integer.m are the implementations of the Integer methods. Also notice how arguments to functions are represented after a colon. One other small difference with C++ is that Objective-C provides the import preprocessor, which acts like the include directive except it includes the file only once.

```
 #import "Integer.h"
 @interface Integer (Display)
 - (id) showint;
 @end
```

Another example follows.

```
#include <stdio.h>
#import "Display.h"

 @implementation Integer (Display)
 - (id) showint
 {
    printf("%d\n", [self integer]);
    return self;
 }
 @end
```

In the second file, we see the first call of an object's method. [self integer] is an example of the way methods are called in Objective-C. This is roughly equivalent to self.integer() in C++. Here are two more, slightly more complicated files:

```
 #import "Integer.h"
 @interface Integer (Add_Mult)
 - (id) add_mult: (Integer *) addend with_multiplier: (int) mult;
 @end
```

and

```
#import "Add_Mult.h"

 @implementation Integer (Add_Mult)
 - (id) add_mult: (Integer *) addend with_multiplier:(int)mult
 {
    return [self set_integer: [self get_integer] + [addend get_integer]
* mult ];
}
@end
```

These two files show how multiple parameters are passed to a function. A label, in this case `with_multiplier`, can be added to the additional parameters. The method is referred to as `add_mult:with_multiplier:`. The following code shows how to call a function requiring multiple parameters.

```
#include <stdio.h>
#import "Integer.h"
#import "Add_Mult.h"
#import "Display.h"

 int main(int argc, char *argv[])
 {
    Integer *num1 = [Integer new], *num2 = [Integer new];
    [num1 integer:atoi(argv[1])];
    [num2 integer:atoi(argv[2])];
    [num1 add_mult:num2 with_multiplier: 2];
    [num1 showint];
 }
```

Building this is as easy as invoking gcc with an additional argument.

```
$ gcc -g -x objective-c main.m Integer.m Add_Mult.m Display.m -lobjc
```

Running the program shows that it can indeed add a number multiplied by two.

```
$ ./a.out 1 4
9
```

As a sample of things to come, consider the disassembled version of the `add_mult:with_multiplier:` function.

```
0x1f02  push   ebp
0x1f03  mov    ebp,esp
0x1f05  push   edi
0x1f06  push   esi
0x1f07  push   ebx
0x1f08  sub    esp,0x1c
0x1f0b  call   0x1f10
0x1f10  pop    ebx
0x1f11  mov    edi,DWORD PTR [ebp+0x8]
0x1f14  mov    edx,DWORD PTR [ebp+0x8]
0x1f17  lea    eax,[ebx+0x1100]
0x1f1d  mov    eax,DWORD PTR [eax]
0x1f1f  mov    DWORD PTR [esp+0x4],eax
0x1f23  mov    DWORD PTR [esp],edx
0x1f26  call   0x400a <dyld_stub_objc_msgSend>
0x1f2b  mov    esi,eax
```