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.

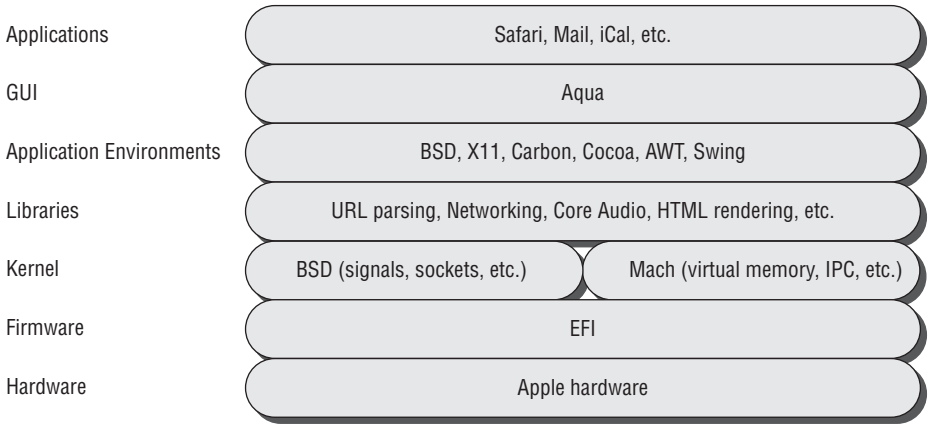

Figure 1-1: Basic architecture of a Mac OS X system

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
```

```
0x1f2d  mov    edx,DWORD PTR [ebp+0x10]
0x1f30  lea    eax,[ebx+0x1100]
0x1f36  mov    eax,DWORD PTR [eax]
0x1f38  mov    DWORD PTR [esp+0x4],eax
0x1f3c  mov    DWORD PTR [esp],edx
0x1f3f  call   0x400a <dyld_stub_objc_msgSend>
0x1f44  imul   eax,DWORD PTR [ebp+0x14]
0x1f48  lea    edx,[esi+eax]
0x1f4b  lea    eax,[ebx+0x10f8]
0x1f51  mov    eax,DWORD PTR [eax]
0x1f53  mov    DWORD PTR [esp+0x8],edx
0x1f57  mov    DWORD PTR [esp+0x4],eax
0x1f5b  mov    DWORD PTR [esp],edi
0x1f5e  call   0x400a <dyld_stub_objc_msgSend>
0x1f63  add    esp,0x1c
0x1f66  pop    ebx
0x1f67  pop    esi
0x1f68  pop    edi
0x1f69  leave
0x1f6a  ret
```

Looking at this, it is tough to imagine what this function does. While there is an instruction for the multiplication (imul), there is no addition occurring. You'll also see that, typical of an Objective-C binary, almost every function call is to `objc_msgSend`, which can make it difficult to know what is going on. There is also the strange call instruction at address 0×1f0b which calls the next instruction. These problems (along with some solutions) will be addressed in more detail in Chapter 6, "Reverse Engineering."

Universal Binaries and the Mach-O File Format

Applications and libraries in Mac OS X use the Mach-O (Mach object) file format and may come ready for different architectures, which are called universal binaries.

Universal Binaries

For legacy support, many binaries in Leopard are *universal binaries*. A universal binary can support multiple architectures in the same file. For Mac OS X, this is usually PowerPC and x86.

```
$ file /bin/ls
/bin/ls: Mach-O universal binary with 2 architectures
/bin/ls (for architecture i386):      Mach-O executable i386
/bin/ls (for architecture ppc7400):   Mach-O executable ppc
```

Each universal binary has the code necessary to run on any of the architectures it supports. The same exact `ls` binary from the code example can run on a Mac with an x86 processor or a PowerPC processor. The obvious drawback is file size, of course. The gcc compiler in Mac OS X emits Mach-O-format binaries by default. To build a universal binary, one additional flag must be passed to specify the target architectures desired. In the following example, a universal binary for the x86 and PowerPC architectures is created.

```
$ gcc -arch ppc -arch i386 -o test-universal test.c
$ file test-universal
test-universal: Mach-O universal binary with 2 architectures
test-universal (for architecture ppc7400):   Mach-O executable ppc
test-universal (for architecture i386):      Mach-O executable i386
```

To see the file-size difference, compare this binary to the single-architecture version:

```
-rwxr-xr-x  1 user1    user1    12564 May  1 12:55 test
-rwxr-xr-x  1 user1    user1    28948 May  1 12:54 test-universal
```

Mach-O File Format

This file format supports both statically and dynamically linked executables. The basic structure contains three regions: the header, the load commands, and the actual data.

The header contains basic information about the file, such as magic bytes to identify it as a Mach-O file and information about the target architecture. The following is the structure from the header, compliments of the /usr/include/mach-o/loader.h file.

```
struct mach_header{

   uint32_t magic;
   cpu_type_t cputype;
   cpu_subtype_t cpusubtype;
   uint32_t filetype;
   uint32_t ncmds;
   uint32_t sizeofcmds;
   uint32_t flags;
};
```

The magic number identifies the file as Mach-O. The cputype will probably be either PowerPC or I386. The cpusubtype can specify specific models of CPU on which to run. The filetype indicates the usage and alignment for the file.

The ncmds and sizeofcmds have to do with the load commands, which will be discussed shortly.

Next is the load-commands region. This specifies the layout of the file in memory. It contains the location of the symbol table, the main thread context at the beginning of execution, and which shared libraries are required.

The heart of the file is the final region, the data, which consists of a number of segments as laid out in the load-commands region. Each segment can contain a number of data sections. Each of these sections contains code or data of one particular type; see Figure 1-2.

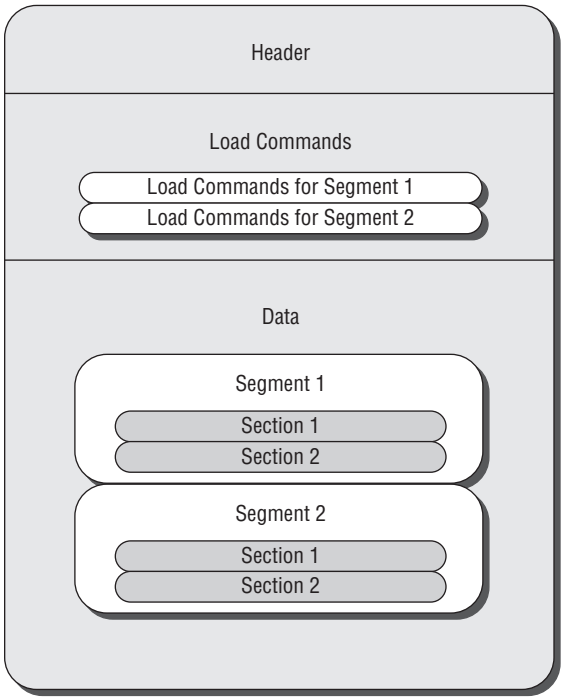

Figure 1-2: A Mach-O file-format example for a file with two segments, each having two sections

Example

All of this information about universal binaries and the Mach-O format is best seen by way of an example. Looking again at the /bin/ls binary, you can see the universal headers using otool.

```
$ otool -f
Fat headers
```

```
fat_magic 0xcafebabe
nfat_arch 2
architecture 0
    cputype 7
    cpusubtype 3
    capabilities 0x0
    offset 4096
    size 36464
    align 2^12 (4096)
architecture 1
    cputype 18
    cpusubtype 10
    capabilities 0x0
    offset 40960
    size 32736
    align 2^12 (4096)
```

Looking at /usr/include/mach/machine.h, you can see that the first architecture has cputype 7, which corresponds to CPU_TYPE_X86 and has a `cpusubtype` of CPU_SUBTYPE_386. Not surprisingly, the second architecture has values CPU_TYPE_POWERPC and CPU_SUBTYPE_POWERPC_7400, respectively.

Next we can obtain the Mach header.

```
$ otool -h /bin/ls
/bin/ls:
Mach header
      magic cputype cpusubtype  caps filetype ncmds sizeofcmds      flags
0xfeedface       7          3  0x00        2    14       1304 0x00000085
```

In this case, we again see the cputype and cpusubtype. The filetype is MH_EXECUTE and there are 14 load commands. The flags work out to be MH_NOUNDEFS | MH_DYLDLINK | MH_TWOLEVEL.

Moving on, we see some of the load commands for this binary.

```
$ otool -l /bin/ls
/bin/ls:
Load command 0
      cmd LC_SEGMENT
  cmdsize 56
  segname __PAGEZERO
   vmaddr 0x00000000
   vmsize 0x00001000
  fileoff 0
 filesize 0
  maxprot 0x00000000
 initprot 0x00000000
   nsects 0
    flags 0x0
Load command 1
```

```
      cmd LC_SEGMENT
  cmdsize 260
  segname __TEXT
   vmaddr 0x00001000
   vmsize 0x00005000
  fileoff 0
 filesize 20480
  maxprot 0x00000007
 initprot 0x00000005
   nsects 3
    flags 0x0
Section
  sectname __text
   segname __TEXT
      addr 0x000023c4
      size 0x000035df
    offset 5060
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x80000400
 reserved1 0
 reserved2 0
…
```

Bundles

In Mac OS X, shared resources are contained in bundles. Many kinds of bundles contain related files, but we'll focus mostly on application and framework bundles. The types of resources contained within a bundle may consist of applications, libraries, images, documentation, header files, etc. Basically, a bundle is a directory structure within the file system. Interestingly, by default this directory looks like a single object in Finder.

```
$ ls -ld iTunes.app
drwxrwxr-x  3 root  admin  102 Apr  4 13:15 iTunes.app
```

This naive view of files can be changed within Finder by selecting Show Package Contents in the Action menu, but you probably use the Terminal application rather than Finder, anyway.

Within application bundles, there is usually a single folder called Contents. We'll give you a quick tour of the QuickTime Player bundle.

```
$ ls /Applications/QuickTime\ Player.app/Contents/
CodeResources       Info.plist     PkgInfo          Resources
Frameworks          MacOS          PlugIns          version.plist
```

The binary itself is within the MacOS directory. If you want to launch the program through the command line or a script, you will likely have to refer to the following binary, for example.

```
$ /Applications/QuickTime\ Player.app/Contents/MacOS/QuickTime\ Player
```

The Resources directory contains much of the noncode, such as images, movies, and icons. The Frameworks directory contains the associated framework bundles, in this case DotMacKit. Finally, there is a number of plist, or property list, files.

Property-list files contain configuration information. A plist file may contain user-specific or system-wide information. Plist files can be either in binary or XML format. The XML versions are relatively straightforward to read. The following is the beginning of the Info.plist file from QuickTime Player.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleDocumentTypes</key>
        <array>
                <dict>
                        <key>CFBundleTypeExtensions</key>
                        <array>
                                <string>aac</string>
                                <string>adts</string>
                        </array>
                        <key>CFBundleTypeMIMETypes</key>
                        <array>
                                <string>audio/aac</string>
                                <string>audio/x-aac</string>
                        </array>
                        <key>CFBundleTypeName</key>
                        <string>Audio-AAC</string>
                        <key>CFBundleTypeRole</key>
                        <string>Viewer</string>
                        <key>NSDocumentClass</key>
                        <string>QTPMovieDocument</string>
                        <key>NSPersistentStoreTypeKey</key>
                        <string>Binary</string>
                </dict>
```

Many of the keys and their meaning can be found at `http://developer.apple.com/documentation/MacOSX/Conceptual/BPRuntimeConfig/Articles/PListKeys.html`. Here is a quick description of those found in the excerpt:

- CFBundleDevelopmentRegion: The native region for the bundle
- CFBundleDocumentTypes: The document types supported by the bundle
- CFBundleTypeExtensions: File extension to associate with this document type
- CFBundleTypeMIMETypes: MIME type name to associate with this document type
- CFBundleTypeName: An abstract (and unique) way to refer to the document type
- CFBundleTypeRole: The application's role with respect to this document type; possibilities are Editor, Viewer, Shell, or None
- NSDocumentClass: Legacy key for Cocoa applications
- NSPersistentStoreTypeKey: The Core Data type

Many of these will be important later, when we're identifying the attack surface in Chapter 3, "Attack Surface." It is possible to convert this XML plist into a binary plist using plutil, or vice versa.

```
$ plutil -convert binary1 -o Binary.Info.plist Info.plist
$ plutil -convert xml1 -o XML.Binary.Info.plist Binary.Info.plist
$ file *Info.plist
Binary.Info.plist:     Apple binary property list
Info.plist:            XML 1.0 document text
XML.Binary.Info.plist: XML 1.0 document text
$ md5sum  XML.Binary.Info.plist Info.plist
de13b98c54a93c052050294d9ca9d119  XML.Binary.Info.plist
de13b98c54a93c052050294d9ca9d119  Info.plist
```

Here we first converted QuickTime Player's Info.plist to binary format. We then converted it back into XML format. The file command shows the conversion has occurred and md5sum confirms that the conversion is precisely reversible.

launchd

Launchd is Apple's replacement for cron, xinetd, init, and others. It was introduced in Mac OS X v10.4 (Tiger) and performs tasks such as initializing systems, running startup programs, etc. It allows processes to be started at various times or when various conditions occur, and ensures that particular processes are always running. It handles daemons at both the system and user level.

The systemwide launchd configuration files are stored in the /System/Library/LaunchAgents and /System/Library/LaunchDaemons directories. User-specific files are in ~/Library/LaunchAgents. The difference between daemons and agents is that daemons run as root and are intended to run in the background. Agents are run with the privileges of a user and may run in the foreground; they can even include a graphical user interface. Launchctl is a command-line application used to load and unload the daemons.

The configuration files for launchd are, not surprisingly, plists. We'll show you how one works. Consider the file com.apple.PreferenceSyncAgent.plist.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.PreferenceSyncAgent</string>
        <key>ProgramArguments</key>
        <array>
                <string>/System/Library/CoreServices/
PreferenceSyncClient.app/Contents/MacOS/PreferenceSyncClient</string>
                <string>--sync</string>
                <string>--periodic</string>
        </array>
        <key>StartInterval</key>
        <integer>3599</integer>
</dict>
</plist>
```

This plist uses three keys. The Label key identifies the job to launchd. ProgramArguments is an array consisting of the application to run as well as any necessary command-line arguments. Finally, StartInterval indicates that this process should be run every 3,599 seconds, or just more than once an hour. Other keys that might be of interest include

- UserName: Indicates the user to run the job as
- OnDemand: Indicates whether to run the job when asked or keep it running all the time
- StartCalendarInterval: Provides cron-like launching of applications at various times

Why should you care about this? Well, there are a few times it might be handy. One is when breaking out of a sandbox, which we'll discuss later in this chapter. Another is in when providing automated processing needed in fuzzing, which we'll discuss more in Chapter 4's section "In-Memory Fuzzing." For example, consider the following plist file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.KeepSafariAlive</string>
        <key>ProgramArguments</key>
        <array>
                <string>/Applications/Safari.app/Contents/MacOS/Safari <
/string>
        </array>
        <key>OnDemand</key>
        <false/>
</dict>
</plist>
```

Save this to a file called ~/Library/LaunchAgents/com.apple.KeepSafariAlive.plist. Then start it up with

```
$ launchctl load Library/LaunchAgents/com.apple.KeepSafariAlive.plist
```

This should start up Safari. Imagine a situation in which fuzzing is occurring while you're using a Meta refresh tag from Safari's default home page. The problem is that when Safari inevitably crashes, the fuzzing will stop. The solution is the preceeding launchd file, which restarts it automatically. Give it a try, and pretend the fuzzing killed Safari.

```
$ killall -9 Safari
```

The launchd agent should respawn Safari automatically. To turn off this launchd job, issue the following command:

```
$ launchctl unload Library/LaunchAgents/com.apple.KeepSafariAlive.plist
```

Leopard Security

Since we're talking about Mac OS X in general, we should talk about security features added to Leopard. This section covers some topics of interest from this field. Some of these address new features of Leopard while others are merely updates to topics relevant to the security of the system.

Library Randomization

There are two steps to attacking an application. The first is to find a vulnerability. The second is to exploit it in a reliable manner. There seems to be no end to vulnerabilities in code. It is very difficult to eliminate all the bugs from an old code base, considering that a vulnerability may present itself as a missing character in one line out of millions of lines of source code. Therefore, many vendors have concluded that vulnerabilities are inevitable, but they can at least make exploitation difficult if not impossible to accomplish.

Beginning with Leopard, one anti-exploitation method Mac OS X employs is library randomization. Leopard randomizes the addresses of most libraries within a process address space. This makes it harder for an attacker to get control, as they can not rely on these addresses being the same. Nevertheless, Leopard still does not randomize many elements of the address space. Therefore we prefer not to use the term *address space layout randomization* (ASLR) when referring to Leopard. In true ASLR, the locations of the executable, libraries, heap, and stack are all randomized. As you'll see shortly, in Leopard only the location of (most of) the libraries is randomized. Unfortunately for Apple, just as one bug is enough to open a system to attacks, leaving anything not randomized is often enough to allow a successful attack, and this will be demonstrated in Chapters 7, 8, and 10. By way of comparison, Windows is often criticized for not forcing third-party applications (such as Java) to build their libraries to be compatible with ASLR. In Leopard, library randomization is not possible even in the Apple binaries!

Leopard's library randomization is not well documented, but critical information on the topic can be found in the /var/db/dyld directory. For example, the map of where different libraries should be loaded is in the dyld_shared_cache_i386.map file in this directory. An example of this file's contents is provided in the code that follows. Obviously, the contents of this file will be different on different systems; however, the contents do not change upon reboot. This file may change when the system is updated. The file is updated when the update_dyld_shared_cache program is run. Since the location in which the libraries are loaded is fixed for extended periods of time for a given system across all processes, the library randomization implemented by Leopard does not help prevent local-privilege escalation attacks.

```
/usr/lib/system/libmathCommon.A.dylib
                  __TEXT 0x945B3000 -> 0x945B8000
                  __DATA 0xA0679000 -> 0xA067A000
               __LINKEDIT 0x9735F000 -> 0x9773D000
/System/Library/Frameworks/Quartz.framework/Versions/
A/Frameworks/ImageKit.framework/Versions/A/ImageKit
                  __TEXT 0x945B8000 -> 0x946F0000
                  __DATA 0xA067A000 -> 0xA0682000
```

```
    __OBJC 0xA0682000 -> 0xA06A6000
  __IMPORT 0xA0A59000 -> 0xA0A5A000
__LINKEDIT 0x9735F000 -> 0x9773D000
```

This excerpt from the dyld_shared_cache_i386.map file shows where two libraries, libmathCommon and ImageKit, will be loaded in memory on this system.

To get a better idea of how Leopard's randomization works (or doesn't), consider the following simple C program.

```
#include <stdio.h>
#include <stdlib.h>

void foo(){
        ;
}

int main(int argc, char *argv[]){
        int y;
        char *x = (char *) malloc(128);
        printf("Lib function: %08x, Heap: %08x, Stack: %08x, Binary:
%08x\n", &malloc, x, &y, &foo);
}
```

This program prints out the address of the malloc() routine located within libSystem. It then prints out the address of a malloced heap buffer, of a stack buffer, and, finally, of a function from the application image. Running this program on one computer (even after reboots) always reveals the same numbers; however, running this program on different machines shows some differences in the output. The following is the output from this program run on five different Leopard computers.

```
Lib function: 920d7795, Heap: 00100120, Stack: bffff768, Binary:
00001f66
Lib function: 9120b795, Heap: 00100120, Stack: bffffab8, Binary:
00001f66
Lib function: 93809795, Heap: 00100120, Stack: bffff9a8, Binary:
00001f66
Lib function: 93d9e795, Heap: 00100120, Stack: bffff8d8, Binary:
00001f66
Lib function: 96841795, Heap: 00100120, Stack: bffffa38, Binary:
00001f66
```

This demonstrates that the addresses to which libraries are loaded are indeed randomized from machine to machine. However, the heap and the application image clearly are not, in this case at least. The small amount of variation in the location of the stack buffer can be attributed to the stack containing

the environment for the program, which will differ depending on the user's configuration. The stack location is not randomized. So while some basic randomization occurs, there are still significant portions of the memory that are not random, and, in fact, are completely predictable. We'll show in Chapters 7 and 8 how to defeat this limited randomization.

Executable Heap

Another approach to making exploitation more difficult is to make it hard to execute injected code within a process—i.e., hard to execute shellcode. To do this, it is important to make as much of the process space nonexecutable as possible. Obviously, some of the space must be executable to run programs, but making the stack and heap nonexecutable can go a long way toward making exploitation difficult. This is the idea behind Data Execution Prevention (DEP) in Windows and W^X in OpenBSD.

Before we dive into an explanation of memory protection in Leopard, we need first to discuss hardware protections. For x86 processors, Apple uses chips from Intel. Intel uses the XD bit, or Execute Disable bit, stored in the page tables to mark areas of memory as nonexecutable. (In AMD processors, this is called the NX bit for No Execute.) Any section of memory with the XD bit set can be used only for reading or writing data; any attempt to execute code from this memory will cause a program crash. In Mac OS X, the XD bit is set on all stack memory, thus preventing execution from the stack. Consider the following program that attempts to execute where the XD bit is set.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char shellcode[] = "\xeb\xfe";

int main(int argc, char *argv[]){
        void (*f)();
        char x[4];
        memcpy(x, shellcode, sizeof(shellcode));
        f = (void (*)()) x;
        f();
}
```

Running this program shows that it crashes when it attemps to exeucte on the stack

```
$ ./stack_executable
Segmentation fault
```

This same program will execute on a Mac running on a PPC chip (although the shellcode will be wrong, of course), since the stack is executable in that architecture.

The stack is in good shape, but what about the heap? A quick look with the vmmap utility shows that the heap is read/write only.

```
==== Writable regions for process 12137
__DATA                 00002000-00003000 [    4K] rw-/rwx SM=COW  foo
__IMPORT               00003000-00004000 [    4K] rwx/rwx SM=COW  foo
MALLOC (freed?)        00006000-00007000 [    4K] rw-/rwx SM=PRV
MALLOC_TINY            00100000-00200000 [ 1024K] rw-/rwx SM=PRV
DefaultMallocZone_0x100000
__DATA                 8fe2e000-8fe30000 [    8K] rw-/rwx SM=COW
/usr/lib/dyld
__DATA                 8fe30000-8fe67000 [  220K] rw-/rwx SM=PRV
/usr/lib/dyld
__DATA                 a052e000-a052f000 [    4K] rw-/rw- SM=COW
/usr/lib/system/libmathCommon.A.dylib
__DATA                 a0550000-a0551000 [    4K] rw-/rw- SM=COW
/usr/lib/libgcc_s.1.dylib
shared pmap            a0600000-a07e5000 [ 1940K] rw-/rwx SM=COW
__DATA                 a07e5000-a083f000 [  360K] rw-/rwx SM=COW
/usr/lib/libSystem.B.dylib
shared pmap            a083f000-a09ac000 [ 1460K] rw-/rwx SM=COW
Stack                  bf800000-bffff000 [ 8188K] rw-/rwx SM=ZER
Stack                  bffff000-c0000000 [    4K] rw-/rwx SM=COW  thread
0
```

Leopard does not set the XD bit on any parts of memory besides the stack. It is unclear if this is a bug, an oversight, or intentional, but even if the software's memory permissions are set to be nonexecutable, *you can still execute anywhere except the stack.* The following simple program illustrates that point.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char shellcode[] = "\xeb\xfe";

int main(int argc, char *argv[]){
        void (*f)();
        char *x = malloc(2);
        memcpy(x, shellcode, sizeof(shellcode));
        f = (void (*)()) x;
        f();
}
```

This program copies some shellcode (in this case a simple infinite loop) onto the heap and then executes it. It runs fine, and with a debugger you can verify that it is indeed executing within the heap buffer. Taking this one step further, we can explicitly set the heap buffer to be nonexecutable and still execute there.

```
#include <sys/mman.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char shellcode[] = "\xeb\xfe";

int main(int argc, char *argv[]){
        void (*f)();
        char *x = malloc(2);
        unsigned int page_start = ((unsigned int) x) & 0xfffff000;
        int ret = mprotect((void *) page_start, 4096, PROT_READ | PROT_
WRITE);
        if(ret<0){ perror("mprotect failed"); }
        memcpy(x, shellcode, sizeof(shellcode));
        f = (void (*)()) x;
        f();
}
```

Amazingly, this code still executes fine. Furthermore, even the stack protections can be overwritten with a call to mprotect.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>

char shellcode[] = "\xeb\xfe";

int main(int argc, char *argv[]){
        void (*f)();
        char x[4];
        memcpy(x, shellcode, sizeof(shellcode));
        f = (void (*)()) x;
        mprotect((void *) 0xbffff000, 4092, PROT_READ | PROT_WRITE |
PROT_EXEC);
        f();
}
```

This might be a possible avenue of attack in a return-to-libc attack. So, to summarize, within Leopard it is possible to execute code anywhere in a process besides the stack. Furthermore, it is possible to execute code on the stack after a call to mprotect.

Stack Protection (propolice)

Although you would think stack overflows are a relic of the past, they do still arise, as you'll see in Chapter 7, "Exploring Stack Overflows." An operating system's designers need to worry about making stack overflows difficult to exploit; otherwise, the exploitation of overflows is entirely trivial and reliable. With this in mind, the GCC compiler that comes with Leopard has an option called `-fstack-protector` that sets a value on the stack, called a canary. This value is randomly set and placed between the stack variables and the stack metadata. Then, before a function returns, the canary value is checked to ensure it hasn't changed. In this way, if a stack buffer overflow were to occur, the important metadata stored on the stack, such as the return address and saved stack pointer, could not be corrupted without first corrupting the canary. This helps protect against simple stack-based overflows. Consider the following program.

```
int main(int argc, char *argv[]){
        char buf[16];
        strcpy(buf, argv[1]);
}
```

This contains an obvious stack-overflow vulnerability. Normal execution causes an exploitable crash.

```
$ gdb ./stack_police
GNU gdb 6.3.50-20050815 (Apple version gdb-768) (Tue Oct  2 04:07:49 UTC
2007)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-apple-darwin"…
No symbol table is loaded.  Use the "file" command.
Reading symbols for shared libraries … done

(gdb) set args
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(gdb) r
Starting program: /Users/cmiller/book/macosx-book/stack_police
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Reading symbols for shared libraries ++. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414141
0x41414141 in ?? ()
(gdb)
```

Compiling with the propolice option, however, prevents exploitation.

```
$ gcc -g  -fstack-protector  -o stack_police stack_police.c
$ ./stack_police AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Abort trap
```

In this case, a SIGABRT signal was sent by the function that checks the canary's value.

This is a good protection against stack-overflow exploitation, but it helps only if it is used. Leopard binaries sometimes use it and sometimes don't. Observe.

```
$ nm QuickTime\ Player  | grep stack
         U ___stack_chk_fail
         U ___stack_chk_guard
$ nm /Applications/Safari.app/Contents/MacOS/Safari | grep stack
```

Here, the nm tool (along with grep) is used to find the symbols utilized in two applications: QuickTime Player and Safari. QuickTime Player contains the symbols that are used to validate the stack, whereas Safari does not. Therefore, the code within the main Safari executable does not have this protection enabled.

It is important to note that when compiling, this stack protection will be used only when the option is used while compiling the specific source file in which the code is located. In other words, within a single application or library, there may be some functions with this protection enabled but others without the protection enabled.

One final note: It is possible to confuse propolice by smashing the stack completely. Consider the previous sample program with 5,000 characters entered as the first argument.

```
(gdb) set args `perl -e 'print "A"x5000'`
(gdb) r
Starting program: /Users/cmiller/book/macosx-book/stack_police `perl -e
'print "A"x5000'`
Reading symbols for shared libraries ++. done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414140
0x920df690 in strlen ()
(gdb) bt
#0  0x920df690 in strlen ()
#1  0x92101927 in strdup ()
#2  0x92103947 in asl_set_query ()
#3  0x9211703e in asl_set ()
#4  0x92130511 in vsyslog ()
#5  0x921303e8 in syslog ()
#6  0x921b3ef1 in __stack_chk_fail ()
#7  0x00001ff7 in main (argc=1094795585, argv=0xbfffcfcc) at
stack_police.c:4
```

The stack-check failure handler, __stack_chk_fail(), calls syslog syslog("error %s", argv[0]);. We have overwritten the argv[0] pointer with our own value. This does not appear to be exploitable, but unexpected behavior in the stack-check failure handler is not a good sign.

Firewall

Theoretically, Leopard offers important security improvements in the form of its firewall. In Tiger the firewall was based on ipfw (IP firewall), the BSD firewall. The ports that are open were controlled by the application's plist files. In Leopard, ipfw is still there but always has a single rule.

```
$ sudo ipfw list
65535 allow ip from any to any
```

Instead the firewall is truly application based and is controlled by /usr/libexec/ApplicationFirewall/socketfilterfw and the associated com.apple.nke.applicationfirewall driver.

Many issues with Leopard's firewall prevent it from being a significant obstacle to attack. The first is that it is not enabled by default. Obviously, if it is not on, it isn't an issue for an attacker. The next is that it blocks only incoming connections. This means any Leopard box that had some services running and listening might be protected; however, out-of-the-box Macs don't have many listening processes running, so this isn't really an issue. If users were to turn on something extra, like file sharing, they would obviously allow connections through the firewall, too. As far as exploit payload goes, it is no more difficult to write a payload that connects out from the compromised host (allowed by the firewall) than to sit and wait for incoming connections (not allowed by the firewall). Regardless, it is hard to imagine a scenario in which the Leopard firewall would actually prevent an otherwise-successful attack from working. Instead, it is basically designed to prevent errant third-party applications from opening listening ports.

Sandboxing (Seatbelt)

Another security feature introduced in Leopard is the idea of sandboxing applications with the kernel extension Seatbelt. This mechanism is based on the principle that your Web browser probably doesn't need to access your address book and your media player probably doesn't need to bind to a port. Seatbelt allows an application developer to explicitly allow or deny an application to perform particular actions. In this way, exploitation of a vulnerability in a particular application doesn't necessarily provide complete access to the system.

Currently the source code for this mechanism is not available, but by looking at and playing around with the XNU source code, it becomes clear how application sandboxing works. The documentation for it is scarce to nonexistent. At this point, this feature is not intended to be used by anyone but Apple engineers, as the following warning indicates.

> *WARNING: The sandbox rule capabilities and syntax used in this file are currently an Apple SPI (System Private Interface) and are subject to change at any time without notice. Apple may in [the] future announce an official public supported sandbox API, but until then Developers are cautioned not to build products that use or depend on the sandbox facilities illustrated here.*

With one exception, applications that are to be sandboxed need to explicitly call the function sandbox_init() to execute within a sandbox. All child processes of a sandboxed function also operate within the sandbox. This allows you to sandbox applications that do not explicitly call sandbox_init() by executing them from within an application in an existing sandbox. One of the parameters to the sandbox_init() function is the name of a profile in which to execute. Available profiles include the following.

- kSBXProfileNoInternet: TCP/IP networking is prohibited.
- kSBXProfileNoNetwork: All sockets-based networking is prohibited.
- kSBXProfileNoWrite: File-system writes are prohibited.
- kSBXProfileNoWriteExceptTemporary: File-system writes are restricted to the temporary folder /var/tmp and the folder specified by the confstr(3) configuration variable _CS_DARWIN_USER_TEMP_DIR.
- kSBXProfilePureComputation: All operating-system services are prohibited.

These profiles are statically compiled into the kernel. We will test some of these profiles in the following code by using the sandbox-exec command. For this command, these profiles are summoned by the terms nointernet, nonet, nowrite, write-tmp-only, and pure-computation.

```
$ sandbox-exec -n nonet /bin/bash
bash-3.2$ ping www.google.com
bash: /sbin/ping: Operation not permitted
bash-3.2$ exit
$ sandbox-exec -n nowrite /bin/bash
bash-3.2$ cat > foo
bash: foo: Operation not permitted
```

Here we demonstrate starting the bash shell with no networking allowed. We omit showing that all the local commands still work and jump straight to trying to use ping, which fails. Exiting out of that sandbox, we try out the nowrite

sandbox and demonstrate that we cannot write files even though normally it would be allowed.

Additionally, it is possible to use a custom-written profile. Although there is no documentation on how to write one of these profiles, there are quite a few well-documented examples in the /usr/share/sandbox directory from which to start. These files are written using syntax from the Scheme programming language and describe all the applications currently sandboxed. These applications include

- krb5kdc
- mDNSResponder
- mdworker
- named
- ntpd
- portmap
- quicklookd
- syslogd
- update
- xgridagentd
- xgridagentd_task_nobody
- xgridagentd_task_somebody
- xgridcontrollerd

Take a look at a couple of these files. The first is quicklookd.

```
;;
;; quicklookd - sandbox profile
;; Copyright (c) 2006-2007 Apple Inc.  All Rights reserved.
;;
;; WARNING: The sandbox rules in this file currently constitute
;; Apple System Private Interface and are subject to change at any time
and
;; without notice. The contents of this file are also auto-generated and
not
;; user editable; it may be overwritten at any time.
;;
(version 1)

(allow default)
(deny network-outbound)
(allow network-outbound (to unix-socket))
(deny network*)

(debug deny)
```

This policy says that, by default, all actions are allowed except those that are specifically denied. In this case, network communication is denied, as the application doesn't need it. Therefore, if this process were taken over by a remote attacker (say, by providing the victim with a malicious file), the process would not be able to open a remote socket back to the attacker. We'll discuss a way around this in a moment.

Another example is update.sb.

```
(version 1)
(debug deny)
(allow process-exec (regex #"^/usr/sbin/update$"))
(allow sysctl-read)
(allow file-read-data file-read-metadata
  (regex #"^/usr/lib/.*\.dylib$"
         #"^/var"
         #"^/private/var/db/dyld/"
         #"^/dev/urandom$"
         #"^/dev/dtracehelper$"))
(deny default)
```

This policy denies all actions by default and allows only those explicitly needed. This is generally a safer approach. In this case, update can read files only from select directories.

Now take a moment to see how this works on a test program. This program takes the name of a file from the command line and attempts to open it, read it, and print the results to the screen; i.e., it is a custom version of the cat utility.

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]){
     int n;
     if(argc != 2){
         printf("./openfile filename\n");
         exit(-1);
       }
       char buf[64];
       FILE *f = fopen(argv[1], "r");
       if(f==NULL){
         perror("Error opening file:");
         exit(-1);
       }
       while(n = fread(buf, 1, 64, f)){
         write(1, buf, n);
       }
       fclose(f);
}
```

Consider the simple policy file. This file allows reading files only from /tmp.

```
(version 1)
(debug deny)
(allow process-exec (regex #"openfile"))
(allow file-read-data file-read-metadata
  (regex #"^/usr/lib/.*\.dylib$"
         #"^/private/tmp" ))
(deny default)
```

We can see this policy being enforced by trying to read a file named hi, which contains only the single word "hi."

```
$ ./openfile hi
hi
$ sandbox-exec -f openfile.sb ./openfile hi
Error opening file:: Permission denied
$ sandbox-exec -f openfile.sb ./openfile /private/tmp/hi
hi
```

Here, the sandbox-exec binary is simply a wrapper that sets the sandbox and then executes the other program within the sandbox as a child. As you can see, the sandbox prevents reading from arbitrary directories, but still allows the application to read from the /tmp directory.

It should be noted that sandboxes are not a cure-all. For instance, in the quicklookd example, network connections are denied but anything else is permitted. One way to achieve network access is to write a file to be executed to the filesystem—perhaps a script that sets up a reverse shell—then configure launchd to start it for you. As launchd is not in the sandbox, there will be no restrictions on this new application. This is one example of circumventing the sandbox.

Additionally, it is difficult to effectively sandbox an application like Safari. This application makes arbitrary connections to the Internet, reads and writes to a variety of files (consider the file:// URI handler as well as the fact a user can use the Save As option from the pull down menu) and executes a variety of applications (through various URI handlers such as ssh://, vnc://, etc). Therefore, it will be hard to write a policy that significantly hinders an attacker who gains control of the Safari process.

One final note is that the Apple-authored software that runs on Windows doesn't have additional security precautions, such as application sandboxing. When you download iTunes for Windows so that you can sync your iPhone, you open yourself up to a remote attack against the mDNSResponder running on your system without its protective sandbox.

References

http://www.matasano.com/log/986/what-weve-since-learned-about-leopard-security-features/

http://www.usefulsecurity.com/2007/11/apple-sandboxes-part-2/

http://developer.apple.com/opensource/index.html

http://www.amazon.com/Mac-OS-Internals-Systems-Approach/dp/0321278542

http://uninformed.org/index.cgi?v=4&a=3&p=17

http://cve.mitre/org/cgi-bin/cvema,e.cgi?name=2006-4392

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3749

http://www.otierney.net/objective-c.html

blog.nearband.com/2007/11/12/first-impressions-of-leopard#