Secret Sauce

How to use or duplicate Apple's private functionality

by Jim Dovey

Copyright ©2010-2011 Jim Dovey. Some Rights Reserved.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. You are free to copy, distribute, and transmit the work, or to adapt the work, provided you adhere to the following conditions:

- **Attribution:** You must attribute the work to Jim Dovey.
- **Noncommercial:** You may not use this work for commercial purposes.
- **Share Alike:** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

Any of the above conditions can be waived by permission of the copyright holder.

This license in no way affects the following rights:

- Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
- The author's moral rights;
- Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

The full text of the copyright license can be found here: http://creativecommons.org/licenses/by-nc-sa/3.0/legalcode

This document was written and typeset using LaTeX 2$_\varepsilon$.

Preface

Apple does some rather cool things in OS X. Many are poorly if at all documented, but can still be duplicated by others, given a little time and effort. Some things we can figure out and implement for ourselves, some work in a way that allows us to hook into them cleanly, and some require the use of more illicit methods, or directly linking against private (and therefore highly mutable) APIs.

This book aims to show the guts of a few elements of OS X which can't be easily reproduced or activated. Some will be complex tasks which require some in-depth knowledge of system behaviour, some will be those which would otherwise require some reverse-engineering know-how to discern. After reading this book, you should be able to implement for yourself some of Apple's 'black magic' features, and should have a good idea of how to go about learning more.

This book is split into five chapters:

Chapter 1 tells you about some of the underlying technologies used in Mac OS X and some of the tools you can use for system and software introspection; it effectively forms the 'starting point' for the other examples. After reading this chapter, you should have a fair understanding of the techniques underlying the discovery and implementation of the following examples.

Chapter 2 starts the exposition with a bang, taking us to the lowest levels of the operating system to investigate how the crash reporter works. After reading this chapter you will be able to write a working crash reporter of your own, or possibly even replace the default system reporter.

Chapter 3 introduces the OS X Managed Client system, *MCX*, known under the moniker of 'parental control' on OS X desktops. After reading this item you will be able to take advantage of MCX at a local level (i.e. without requiring a copy of OS X Server) and will be able to integrate some of its functions into your own applications.

Chapter 4 covers the guts of the authentication system, in particular the means in which login and fast-user-switching are implemented. After reading this chapter you should be able to create advanced multi-stage authorization tasks with their own user interfaces; you may even be prepared to take on the task of replacing the OS X login window altogether...

Chapter 5 closes out the book with information on one of the juiciest private APIs in OS X: the *Time Machine* user interface. After reading this chapter, you should be able to have your application's user interface interact directly with the Time Machine system in a manner similar to the iPhoto or Address Book applications.

The book is designed such that each chapter functions as a self-contained tutorial, so you are free to skip directly to the topic that interests you most. However, it is recommended that you begin with Chapter 1 in order to get a thorough grounding in some of the underlying system details. In many cases, it was this knowledge which enabled me to search out much of the rest of the content of this book.

Each topic will present some information on standard system components in a general manner, in order to educate about the workings of the system. A goal of this approach is that you should know enough to understand the nuances of the topic enough to make discoveries or uses of your own. Once that is done, I will show how to make use of that information toward the specific aim of the chapter or section.

If there is some task or subsystem in particular that you would like to see explored in future revisions of this book, please let me know. The current content is based on my own rather focussed experience, which for instance does not involve a lot of user-facing applications. I would be very interested to find out just what sort of itches people need to scratch, and which of Apple's fancy tricks they would like to replicate, and I am always on the lookout for new challenges. Also, I may end up making additions myself,

for example there may in future be a description of AppleTV API hacking and the like, which I'm sure would be interesting to one or two people out there...

Jim Dovey <jimdovey@mac.com>

Toronto, ON
Canada
December 11, 2010

Contents

Chapter 1

Groundwork

The first part of this chapter will introduce the messaging subsystem used by a great many of OS X's programming interfaces. The second part will look at the way user profile data is handled, and the third will explain the reasoning behind and facilities of OS X's authentication system. Together these will provide a good foundation for the topics discussed in the remainder of the book.

1.1 Mach Messaging

A lot of the APIs in OS X make use of the Mach messaging subsystem. This is a low-latency, low-memory, kernel-based IPC mechanism, and is a core component of the Mach operating system (upon which OS X's xnu kernel is based). In most cases, these APIs are generated by the Mach Interface Generator (*MIG*) tool based on some easy-to-write interface definitions. One example is the CoreGraphics API: a great many of the methods in the `CGxxxxx.h` header files are actually MIG-generated stubs which simply pack up their arguments and send them to the *WindowServer* process. Resize a window in a web browser and you'll trigger a massive number of Mach IPC calls— this should provide a good example of how fast the Mach messaging subsystem can be.

1.1.1 Ports

Mach messaging, and indeed much of the Mach microkernel (and therefore much of Apple's *XNU* kernel), revolves around the concept of *ports*. A port

is a communication channel which is accessible through the acquisition of discrete send and receive *rights*, and allows the transfer of *messages* as typed data structures.

A port is itself a unidirectional communication channel in a client/server model. A port usually has a single receiver and potentially many senders; each client wishing to access a service would obtain a *send-right* to the server, which would enqueue messages onto the server's *receive-right*. If it sounds as though *port* and *right* are being conflated here, well, that's correct. In Mach parlance, so many things rely on ports that it behooves us to use slightly different terminology to distinguish the many appearances of the type. In this case a *send-right* refers to a port which is used to *send* a message, while a *receive-right* is a port used to *receive* those messages. We will see later how the two are made to fit together.

Another piece of terminology used to refer to ports (and which we'll see shortly) is a *name*. Since most system services and entities under Mach are referenced and named by a port, this port is often referred to as that entity's *name*.

A port is represented by a 32-bit numeric value. Part of this value is a bitfield used to identify the type of resource to which the port refers (is it a client/server IPC endpoint, is it a task, a thread, etc.). The remainder can be thought of as a resource index similar to file descriptors under UNIX. This isn't necessarily true, mind you, but this is conceptually what we're dealing with.

Ports are managed and allocated by the kernel, and the main state associated with them is their *message queue*. A port also maintains a count of references/rights to it. This means that ports are reference-counted in the manner we're used to seeing with CoreFoundation and Cocoa; however, it is quite possible (indeed encouraged, under certain circumstances) to destroy a port outright without regard to its reference count. This is safe, since any interested parties can request a notification from the kernel when a given port is destroyed or otherwise becomes inoperative— in the Mach world, this state is called *port death*.

Ports also exist within a per-task *name space*. That means that when you obtain a send right to a server, the port value you receive is not unique in the system, it is only unique to your process. However, the Mach kernel provides facilities to move ports between different name spaces (tasks) with relative ease. The *port name* held by two discrete tasks might have different numerical

values, but they will refer to the same underlying port, and therefore will operate upon the same message queue.

The messages sent using Mach ports are typed data collections. They are not system objects in their own right, but since they are queued they are designed to be able to hold state between the time a message is sent and the time it is received. A message can contain not only pure data copied between address spaces by the kernel, but can also contain references to virtual memory ranges and port rights. Thus is it possible to send a very large chunk of data to a server process as a virtual memory range, and provide a reply port along with it to which the server can send a completion message. Using virtual memory means that, rather than copying the data, the physical memory underlying the specified virtual range is simply mapped into the receiving task's address space. This makes the transfer of large amounts of data using Mach ports quite fast and resource-light.

1.1.2 Accessing Services

As a client, you need some means of locating a service with which you intend to communicate. Since, as we've already discussed, the numerical values representing ports are unique to each task (process) running on the system, we require some other means of advertising and locating the ports vended by such services. Luckily for us, we have the *bootstrap server* to help us out.

The bootstrap server has a simple interface, defined in `<servers/bootstrap.h>`. From Mac OS X 10.4 onwards, it is implemented as part of *launchd* (which we will cover later in this chapter). Prior to this it was part of the *mach_init* process.

Let's look at the bootstrap client-side API for a moment (see Listing 1.1).

Listing 1.1: Bootstrap Client API

```
1 /*
2  * bootstrap_look_up()
3  *
4  * Returns a send right for the service port
5  * declared/registered under the name service_name.
6  * The service is not guaranteed to be active.  Use
7  * the bootstrap_status call to determine the status
8  * of the service.
9  *
```

```
 * Errors:  Returns appropriate kernel errors on rpc
 * failure.
 *   Returns BOOTSTRAP_UNKNOWN_SERVICE, if service does
 * not exist.
 */
kern_return_t bootstrap_look_up(
        mach_port_t bp,
        const name_t service_name,
        mach_port_t *sp);
```

Yes, that's the entire thing— everything else in that file is for servers' use, and most of *that* is actually deprecated now.

To resolve a service, you pass in a pointer to an uninitialized `mach_port_t` in the last parameter, and if the lookup succeeds, a send right to the server's port is returned by reference. The second parameter is a C-string of the service's vended port name. The first is the bootstrap port, which is set up by *launchd* and passed on to new processes as they are launched. To access this value, just reference the `bootstrap_port` external variable.

It's common for a server to vend its service port using a reverse-DNS styled name, such as `com.apple.DirectoryService`. To look up that port you would use the code in Listing 1.2.

Listing 1.2: Locating a Service Using the Bootstrap Server

```
mach_port_t service = MACH_PORT_NULL;
kern_return_t kr = bootstrap_look_up(bootstrap_port, "↩
      com.apple.DirectoryService", &service);

/* use the port */
/* ... */

/* when done, release your reference to it */
mach_port_deallocate( mach_task_self(), service );
```

1.1.3 The Mach Interface Generator

The most common means of vending Mach services (and the way which the bootstrap server itself is vended) is using the *Mach Interface Generator* tool. This enables you to create interface definition files specifying the types and methods you want to export using a simplified syntax, and the *mig* tool will

generate C source and header files for client, server, or both. An example of the syntax can be seen in Listing 1.3 below.

Listing 1.3: An Example MIG IPC Definition

```
1  subsystem FCSHelperMIG 75000;
2
3  userprefix fcsmig_;
4  serverprefix fcsmig_do_;
5
6  #include <mach/std_types.defs>
7  #include <mach/mach_types.defs>
8
9  import "FCSHelperMIG_types.h";
10 import <Security/Authorization.h>;
11
12 type authInfo_t = array [32] of char;
13 type migString_t = c_string [*:512];
14 type propList_t = array [*:1024] of char;
15 type xmlData_t = array [] of char
16                  ctype: vm_address_t;
17
18 routine get_output_from_command
19 (
20         helper              : mach_port_t;
21         authorization       : authInfo_t;
22         arguments           : propList_t;
23         arguments_ool       : pointer_t, Dealloc;
24     out xml_data            : xmlData_t;
25     out xml_data_ool        : pointer_t, Dealloc
26 );
27
28 routine run_basic_command
29 (
30         helper              : mach_port_t;
31         authorization       : authInfo_t;
32         arguments           : propList_t;
33         arguments_ool       : pointer_t, Dealloc
34 );
35
36 simpleroutine set_log_level
37 (
38         helper              : mach_port_t;
39         level               : int32_t
```

```
40 );
```

Let's go through the contents of this file in sequence:

- **Line 1:**
  The `subsystem` directive associates a name and a numeric prefix value with the service. Each routine defined in a given subsystem is assigned a numeric value to distinguish it from other routines, and these numbers begin at the subsystem value specified here.

- **Lines 3-4:**
  The `prefix` values specify a prefix for the C function names generated by the MIG tool. For a routine named `function`, on the client (user) side the MIG tool will create a C function titled `fcsmig_function`. On the server side it would generate code which calls out to a function you would define called `fcsmig_do_function` to perform the requested function.

- **Lines 6-7:**
  `#include` works in MIG in exactly the same way as in C: it reads the specified source file into the current file at this location.

- **Lines 9-10:**
  The `import` directive is placed directly into the generated C header files as a C `#include` directive. The trailing semicolon is not emitted in this case.

- **Lines 12-16:**
  Types can be defined in MIG in a similar manner to a C `typedef`. In this case, we define a number of types as essentially arrays of bytes. The `array` keyword is used to define an array with an optional size component, specified within the square braces. If the braces are empty, the array size is unspecified (and effectively unbounded). If a constant value is there, then the array *always* contains that many elements. In the case of something like `[*:512]`, the array can be any size up to a maximum of 512 elements.
  On Line 16 you can see the `ctype` keyword. This is used to specify a different type name to use in the generated C header and source files.

- **Lines 18-40:**
  These lines contain the IPC function definitions themselves. There are

two types available in MIG: the *Routine* and the *SimpleRoutine*. A *Routine* is a method which will not return control to the client until the server has completed its task. It is used for IPC messages which require some form of reply. A *SimpleRoutine* is a basic one-way message which returns immediately; as such, there is no guarantee that the server has processed the message, only that it was added to the server's message queue.
Within the routine definitions are the parameters. In MIG, there is a required (but not implicit) first argument of `mach_port_t`. This is the server's port. The real arguments follow. They are always stated as *identifier* `:` *type*, and each parameter must be separated by semicolons. There are some additional keywords you can see in the example. The `out` keyword specifies that the given parameter is used solely for a return-by-reference result (a corresponding `in` keyword is implicit). There is also an `inout` keyword, referring to a return-by-reference value which also contains a starting value sent to the server. An example would be a count value, where the caller specifies the size it has allocated and the server modifies it to specify the amount used, or the amount required.
The last keyword we can see in the example is the `Dealloc` keyword following some parameter types of `pointer_t`. This specifies that the given parameter refers to memory which should be deallocated using the `vm_deallocate` function. On input parameters, this deallocation is taken care of by the MIG-generated stub code. On output parameters it is a hint to the client. It also tells both sides of the MIG stub code to allocate memory locally to hold the value, rather than include it inline similar to a MIG `array` type.

The mig command-line tool uses the definition file to generate a client-side header file and both client and server-side implementation files. The header file contains a specific structure and keywords which clearly identify it as having been generated by MIG. For example, the generated C function declaration for the `run_basic_command` routine defined in Listing **??** above can be seen in Listing 1.4.

Listing 1.4: The Generated C Function

```
1 /* Routine run_basic_command */
2 #ifdef  mig_external
3 mig_external
```

```
4  #else
5  extern
6  #endif  /* mig_external */
7  kern_return_t fcsmig_run_basic_command
8  (
9      mach_port_t helper,
10     authInfo_t authorization,
11     propList_t arguments,
12     mach_msg_type_number_t argumentsCnt,
13     vm_offset_t arguments_ool,
14     mach_msg_type_number_t arguments_oolCnt
15 );
```

We can see immediately that there is a `mig_external` definition at the top of the function, and the comment above that specifies whether the source routine definition was for a Routine or a SimpleRoutine. Take note of these clues, because you can infer a lot about function types and arguments using this information.

Looking further down the file, we see some structure definitions, including the one in Listing 1.5.

Listing 1.5: The Message Structure

```
1  #ifdef  __MigPackStructs
2  #pragma pack(4)
3  #endif
4      typedef struct {
5          mach_msg_header_t Head;
6          /* start of the kernel processed data */
7          mach_msg_body_t msgh_body;
8          mach_msg_ool_descriptor_t arguments_ool;
9          /* end of the kernel processed data */
10         NDR_record_t NDR;
11         authInfo_t authorization;
12         mach_msg_type_number_t argumentsCnt;
13         char arguments[1024];
14         mach_msg_type_number_t arguments_oolCnt;
15     } __Request__run_basic_command_t;
16 #ifdef  __MigPackStructs
17 #pragma pack()
18 #endif
```

Here we can see the layout of the actual message structure that is being sent out to the server. After looking at this and more, you will notice that

it always consists of a `mach_msg_header_t` followed by a `mach_msg_body_t` and a `mach_msg_ool_descriptor_t`. After this is an `NDR_Record_t`, and what follows maps to the routines arguments. Note that the variably-sized data arguments have been split into a buffer and a count, both here and in the C function in Listing 1.4.

After a little checking of other structures, we can determine that the `mach_msg_ool_descriptor_t` is only included under certain circumstances, namely when an unbounded array is being passed (it'll be sent inline if under a certain size, and sent out-of-line if larger). But when it's included, this is where it's placed. So we have two possible structure types. Great— two types we can work with.

The last piece of knowledge that we need is how these messages are handled server-side. We already know that, given a routine name of `function` and a userprefix of `go_`, the generated client-side function will be named `go_function`. On the server side, we've specified a serverprefix of `do_`, which leads to a server method called `do_function`. But MIG doesn't implement that for us, it builds code which calls it— we have to implement it ourselves. So what does MIG use to handle the message receipt and marshalling? It has its own internal prefix, `_X`. So on the server side, in addition to the real IPC endpoint of `do_function()` there is also a message-marshalling routine called `_Xfunction()`. This function will always adhere to a simple standard: it will return `kern_return_t`, and it will receive two arguments, both of which are pointers to mach messages. The first argument is the incoming message, the second is the outgoing one to fill out. And since we just saw what these look like, we know what's going to be inside them.

1.1.4 Putting The Knowledge To Use

You will be forgiven for thinking that the above is all very much academic, but let's consider a real-world example, albeit one based around quite a narrowly-defined scenario, where this knowledge is absolutely invaluable. This scenario revolves around the Application Enhancer package from Unsanity.

The Application Enhancer implements a system-wide means of loading third-party code into any process on a per-user-session basis. It does this by injecting code into the WindowServer process at user login and patching a core function used by all other GUI-based applications to check into the UI system (something which is required in order to receive Carbon/Cocoa events and to use the window manager). This core function is called synchronously

by the other processes in their main threads, and there is no timeout (at least, none that I've seen so far). Their patch runs before this method can complete, which means that it can start copying code and data into the calling application's address space without worrying that the application's main thread will get in the way.

The function in question is an internal routine called `_CGSCheckInApplication`. Most of the WindowServer application is actually implemented in the Core-Graphics framework, so we'll use the *nm* tool to dump a list of all the available functions, and we'll filter the output using Grep. The output is shown in Listing 1.6.

Listing 1.6: The `_CGSCheckInApplication` routine and friends.

```
1 0000000000000557d t __CGSCheckInApplication
2 00000000001122e3 t __CGXCheckInApplication
3 0000000000112143 t __XCheckInApplication
```

Here we can see three variants of the method. One beginning with `_CGS`, one beginning with `_CGX`, and one beginning with `_X`. This looks familiar, no? Quite right, it's a MIG function. Evidently the WindowServer interface is built using MIG, which means we are able to make a number of assertions:

1. We know exactly what the last few stack frames of the calling thread will look like: it's called `_CGSCheckInApplication()`, which has called `mach_msg_overwrite()` to deliver the message and wait for the result, and *that* calls `mach_msg_overwrite_trap()` to actually trap into the kernel code and suspend the thread until the reply is ready. This means that we can make changes to that thread to modify the return address of, say, the `mach_msg_overwrite()` frame to call some injected code. This injected code can then load an external library to setup the Application Enhancer subsystems properly and cleanly. This is safe because we can guarantee that the stack is in this known-good state for the operation, and we know that it won't change while we're trying to make our own edits.

2. We know exactly the arguments coming into the server function `_XCheckInApplication()`. This means that we aren't working blind when we patch that function— we know exactly what registers to save, what part of the stack frame to save, and so on.

Now, the actual structure of the message sent to `_XCheckInApplication()` isn't known to us, but we *do* know what it will begin with, so we know where to start looking within the message structure. With a bit of foresight, we should be able to determine some useful information from what we see on the stack there. And of course it's easy to attach `gdb` to the WindowServer to try this out— just be sure to do it via an ssh session, since your UI will freeze as soon as the WindowServer hits a breakpoint...

In my case, after a little poking around, I was able to work out where in the structure the calling application's process ID was located, and also where its *flavor* was to be found (the flavor variable indicates whether the app is native, a PowerPC binary running on Intel via Rosetta, or a Classic application). This was all I needed to make a decision on whether to install patches in that application. The code in Listing 1.7 shows the patch routine I used in my own Application-Enhancer-like application. The actual patching and injection implementation... well, that's for another day.

Listing 1.7: A Patch On Both Your Houses

```
1  static kern_return_t __checkin_app_patch(↩
       mach_msg_header_t *InHeadP, mach_msg_header_t *↩
       OutHeadP)
2  {
3  #ifdef __MigPackStructs
4  #pragma pack(4)
5  #endif
6      typedef struct {
7          mach_msg_header_t Head;
8          NDR_record_t NDR;
9          ProcessSerialNumber procPSN;
10         uint32_t            __off40;
11         uint32_t            __off44;
12         uint32_t            flavor;
13         pid_t               proc_id;
14     } Request;
15 #ifdef __MigPackStructs
16 #pragma pack()
17 #endif
18
19     pthread_mutex_lock( &csx_mutex );
20
21     // don't do anything if our management daemon
22     //  isn't actually running
```

```
23      if ( cmx_port != MACH_PORT_NULL )
24      {
25          Request *pIn = (Request *) InHeadP;
26
27          debug_log( "Received application startup msg, ↩
                cmx_port = %#x", cmx_port );
28
29          pid_t procid = pIn->proc_id;
30          uint32_t flavor = pIn->flavor;
31          boolean_t rosetta = FALSE;
32
33          // check to see if we need to byte-swap things
34          if ( pIn->NDR.int_rep != NDR_record.int_rep )
35          {
36              procid = OSSwapInt32(procid);
37              flavor = OSSwapInt32(flavor);
38  #if defined(__i386__) || defined(__x86_64__)
39              rosetta = TRUE;
40  #endif
41          }
42
43          // don't do anything if the flavor is '1',
44          // because that means it's a Classic process
45          // also don't patch unless the app's process
46          // group ID is equal to our own process ID
47          // i.e. avoid apps not launched by the
48          // WindowServer
49          if ( (flavor != 1) && (getpgid(procid) == ↩
                getpid()) )
50          {
51              __checkin_core( procid, rosetta );
52          }
53      }
54
55      pthread_mutex_unlock( &csx_mutex );
56
57      return ( __checkin_app_orig(InHeadP, OutHeadP) );
58  }
```

1.2 User Profile Data

1.2.1 Directory Services

There are a few kinds of user data in Mac OS X. The first is the most obvious, namely the files which exist within a user's home folder, and includes their documents, images, and the like. The second also resides in the user's home, although only as an implementation detail: the user's preferences for interacting with the system and applications. The last resides in a central database accessible (to some degree) by virtually anyone on the computer. This last is the user profile, and supports their ability to sign into the computer, to set their profile picture, and change their password. These records are also the source for more esoteric information, such as that related to a user's UNIX group membership and filesystem privileges.

On OS X, this data is stored in a series of database files in a hidden directory (if you're feeling adventurous, look inside `/var/db/dslocal`). In earlier versions of OS X it was stored in a similar location, but was part of the venerable *NetInfo* system which dated back to *NeXTstep*.

The facility by which this data is managed and accessed is called *Open Directory*. This is Apple's implementation of the standard LDAP form of directory-based data management. Open Directory is also the name for the family of APIs used to access this data, at the core of which sits `DirServices.framework`. The setup of this service on Mac OS X is that a single daemon provides the public API and request marshalling service, while various plugins are loaded which are able to access different forms of data stores. One such store was NetInfo, now supplanted by the Local store described above. Other stores include Microsoft's *Active Directory* and standard LDAP services (utilizing either v2 or v3 of the protocol). There is additionally a plugin which supports obtaining information from standard BSD configuration files.

The directory server crucially maintains a *search list* of plugins (called *nodes* in DS parlance), which is editable by any user with administrative rights through the Directory Utility application. In OS X 10.5 this was located in the Utilities folder, but in 10.6 it was moved into `/System/Library/CoreServices`. This search list defines the order in which plugins are asked to fulfil an information request from a client. An additional search list exists specifically for contact information lookup, which means that you can for instance hook contact lookup into an external LDAP server but look for configuration details only on the local system.

1.2.2 Exploring Open Directory

We can explore the contents of our Open Directory setup using the `dscl` tool. This operates in both an interactive mode and a straightforward command-line invocation, receiving options and commands from the command-line and sending data to standard output before terminating. In interactive mode, a number of commands are available, including an `ls` command to list the contents of the current directory node, as shown in Listing 1.8.

Listing 1.8: Listing Open Directory Nodes

```
% dscl localhost
 > ls
BSD
Local

Contact
Search
 > cd Local/Default
/Local/Default >
```

The default store is located at the path `/Local/Default`, and within it are listed all the known record types. A small sample is shown in Listing 1.9.

Listing 1.9: Listing Open Directory Record Types

```
/Local/Default > ls
...
Services
SharePoints
SMBServer
Users
WebServer
/Local/Default >
```

A particularly interesting record type is `SharePoints`, which is used to store the details of any folders you share (or which are shared automatically by the system). To see what records of a particular type are available, you can use `ls` to read that type as if it were a directory. And then to read the data stored in a record you would use the `read` command. An example of its output from a SharePoint record can be seen in Listing 1.10, where you can see information used for sharing the item under both the AFP and SMB file-server protocols.

Listing 1.10: An Open Directory SharePoint Record

```
/Local/Default > read SharePoints/Jim\ Dovey's\ Public\↩
      Folder/
dsAttrTypeNative:afp_guestaccess: 1
dsAttrTypeNative:afp_name:
 Jim Dovey's Public Folder
dsAttrTypeNative:afp_shared: 1
dsAttrTypeNative:directory_path: /Users/jim/Public
dsAttrTypeNative:ftp_name:
 Jim Dovey's Public Folder
dsAttrTypeNative:sharepoint_account_uuid: 0F981803↩
     -0568-490D-BCD9-E4C49CDA5774
dsAttrTypeNative:sharepoint_group_id: 69CBFCCA-595B-4↩
     CE1-A82F-F29AD0EE0C6C
dsAttrTypeNative:smb_createmask: 644
dsAttrTypeNative:smb_directorymask: 755
dsAttrTypeNative:smb_guestaccess: 1
dsAttrTypeNative:smb_name:
 Jim Dovey's Public Folder
dsAttrTypeNative:smb_shared: 1
AppleMetaNodeLocation: /Local/Default
RecordName:
 Jim Dovey's Public Folder
RecordType: dsRecTypeStandard:SharePoints
/Local/Default >
```

There are a lot of other types to look at, but the most interesting items are the `Computers`, `ComputerGroups`, `Groups`, and `Users` types. And even more so when you look at the record for a User for whom Parental Controls are enabled. We'll see more of that later, in Chapter 3. Listing 1.11 shows the record of a standard user account with no special features.

Listing 1.11: An Open Directory User Record

```
/Local/Default > read Users/bob/
AppleMetaNodeLocation: /Local/Default
AuthenticationAuthority: ;ShadowHash; ;Kerberosv5;;↩
     bob@LKDC:SHA1.21↩
     CD166FEFC128883FCB504AC38FAEF2E40CEA42;LKDC:SHA1↩
     .21CD166FEFC128883FCB504AC38FAEF2E40CEA42;
AuthenticationHint: test
GeneratedUID: 4ED7E269-B428-4482-B01B-DC11B376966C
NFSHomeDirectory: /Users/bob
Password: ********
```

```
Picture:
 /Library/User Pictures/Instruments/Drum.tif
PrimaryGroupID: 20
RealName:
 Bob McTesting-Person
RecordName: bob
RecordType: dsRecTypeStandard:Users
UniqueID: 502
UserShell: /bin/zsh
```

For now, we've seen enough of the workings of Open Directory to be able to look deeper into the actual data stored here without undue head-scratching later on. This means that it's time to step on to our next subject.

1.3 Authorization

1.3.1 Authorization vs. Authentication

Though the two are often confused, the tasks of *authentication* and *authorization* are quite different, and solve different problems.

- **Authentication** refers to the process of determining the identity of some entity, whether it be a human, a computer, or a process. An encrypted connection using certificates or a password dialog are common ways of implementing authentication.
- **Authorization** is the process of obtaining a user's consent for undertaking a particular action, such as making modifications to a system file or some configuration.

A key aspect of authorization is that while it might well involve some form of authentication, this isn't a requirement. For instance, a frequent form of authorization is used when you modify a file: the system checks whether your effective user ID and group ID matches those with permission to modify the file. Another example would involve copying items into the `Applications` folder. If you're logged in as a user with administrative privileges, the copy just happens. If you're not, then a dialog pops up requesting that an administrator enter their name and password to authorize the action.

1.3.2 Obtaining Authorization

The standard way of obtaining user authorization for your actions on OS X is to use the Authorization API from the Security framework. Looking at `<Security.framework/Authorization.h>` shows the main routines involved.

The Authorization API is based around the `AuthorizationRef` pseudo-object type. This object encapsulates a collection of requested *rights*, such as the right to modify a file, or to execute an application with administrative privileges. The code in Listing 1.12 illustrates the process of obtaining a commonly-used authorization right in OS X 10.6.

Listing 1.12: Creating an Authorization

```
1  // start with an empty authorization reference
2  AuthorizationRef authRef = NULL;
3  OSStatus status = noErr;
4  status = AuthorizationCreate( NULL, ↩
       kAuthorizationEmptyEnvironment, ↩
       kAuthorizationFlagDefaults, &authRef );
5  assert(status == errAuthorizationSuccess);
6
7  // the right we're requesting
8  AuthorizationItem right = { ↩
       kSMRightBlessPrivilegedHelper, 0, NULL, 0 };
9  AuthorizationRights rightSet = {1, &right};
10
11 AuthorizationFlags flags = ↩
       kAuthorizationFlagInteractionAllowed | ↩
       kAuthorizationFlagExtendRights;
12
13 // authorize some rights -- will prompt the user
14 status = AuthorizationCopyRights( authRef, &rightSet, ↩
       kAuthorizationEmptyEnvironment, flags, NULL );
15 assert(status == errAuthorizationSuccess);
16
17 // use the authorization...
18 // ...
19
20 AuthorizationFree( authRef, ↩
       kAuthorizationFlagDestroyRights );
```

If you're using Objective-C, you can make use of a slightly less flexible but much simpler to use API in the form of `SecurityFoundation.framework`. The equivalent Objective-C version of the code above is shown in Listing 1.13.

Listing 1.13: Obtaining Authorization in Objective-C

```
1  AuthorizationFlags flags = ↩
       kAuthorizationFlagExtendRights|↩
       kAuthorizationFlagInteractionAllowed;
2  NSError *error = nil;
3  SFAuthorization *auth=[SFAuthorization authorization];
4  if ( [auth obtainWithRight:↩
       kSMRightBlessPrivilegedHelper
5                     flags:flags
6                     error:&error] )
7  {
8      // Use the right
9  }
10 // the authorization object has been autoreleased ↩
       already
```

1.3.3 Behind the Scenes

Sooner or later the question arises: how does the system know how to authorize a particular right? Well, the answer lies in another hidden system folder, albeit one well-known to UNIX folks: `/etc`. There is a property list file there, writable only by root, called simply `authorization`. It is this file which contains all the definitions of what types of rights exist, and how to go about authorizing them. In Listing 1.14 you can see the definition corresponding to the right we requested in the sample code above.

Listing 1.14: The `SMJobBless()` Right Definition

```
1  <key>com.apple.ServiceManagement.blesshelper</key>
2  <dict>
3      <key>class</key>
4      <string>rule</string>
5      <key>comment</key>
6      <string>Used by the ServiceManagement framework to ↩
           add a privileged helper tool to the system ↩
           launchd.</string>
7      <key>k-of-n</key>
```

```
8       <integer>1</integer>
9       <key>rule</key>
10      <array>
11          <string>is-root</string>
12          <string>authenticate-admin-30</string>
13      </array>
14  </dict>
```

Let's look at this and see what it all means. First of all, we see that its **class** is set to **rule**. This tells the system that to acquire this right, one or more of a number of rules must be matched. The **k-of-n** value of **1** says that only one of the specified rules need match for the right to be granted. Lastly comes the list of rules to use:

- **is-root** seems fairly straightforward: if the caller is running as root, then they are automatically considered authorized.
- **authenticate-admin-30** is a little less so. We can infer that it means 'an admin password is required', but it might just check whether the caller is an administrator. And what does the '30' mean?

To investigate the rules, we look further down the same file, where we find all these rules defined for us. In particular, we see `authenticate-admin-30` and a similar-looking item titled `is-admin`. These are shown in Listing 1.15.

Listing 1.15: Administrator Authorization Rules

```
1  <key>authenticate-admin-30</key>
2  <dict>
3      <key>class</key>
4      <string>user</string>
5      <key>comment</key>
6      <string>Like the default rule, but
7      credentials remain valid for only 30 seconds after ↩
           they've
8    been obtained.  An acquired credential is shared by ↩
         all clients.
9      </string>
10     <key>group</key>
11     <string>admin</string>
12     <key>shared</key>
13     <true/>
```

```
14      <key>timeout</key>
15      <integer>30</integer>
16  </dict>
17  <key>is-admin</key>
18  <dict>
19      <key>authenticate-user</key>
20      <false/>
21      <key>class</key>
22      <string>user</string>
23      <key>comment</key>
24      <string>Verify that the user asking for ↩
            authorization is an administrator.</string>
25      <key>group</key>
26      <string>admin</string>
27      <key>shared</key>
28      <string>true</string>
29  </dict>
```

The comments in those entries make clear what they are designed to do. You can see that they have their own **class** entries, which this time are both set to **user**. This means that they will check something to do with the UNIX effective user and/or group IDs of the caller in order to make a decision. We can then see that they both have a **group** key with a value of `admin`. This means that they will both check whether the caller is a member of the admin group. We can also see that they are both **shared** rights available to any application, not just the caller, and we can see the 30-second timeout defined on `authenticate-admin-30`.

There's one other difference worth pointing out though. The `is-admin` right has an extra key: **authenticate-user**, with a value of **false**. This tells the system that for the `is-admin` right, it should *not* request authentication, it should *only* check the caller's group membership. The `authenticate-admin-30` rule however uses the default behaviour, which is to present an authentication dialog to confirm that a member of the admin group is actually present at the console.

Our last stop on this whistle-stop tour of the authorization rules list is the `authenticate` rule, which you can see in Listing 1.16. This has a class of **evaluate-mechanisms**, and a list of mechanisms to evaluate.

Listing 1.16: The `authenticate` Rule

```
1  <key>authenticate</key>
```

```
<dict>
    <key>class</key>
    <string>evaluate-mechanisms</string>
    <key>mechanisms</key>
    <array>
        <string>builtin:smartcard-sniffer,privileged</string>
        <string>builtin:authenticate</string>
        <string>builtin:authenticate,privileged</string>
    </array>
</dict>
```

This looks fairly interesting. Can we change this? Well, yes. We only need to replace the mechanisms listed with some of our own and we can become the ultimate authority on the system. How's that for *power*?

As to what exactly authorization mechanisms are and how we create them, we'll investigate this further in Chapter 4.

1.4 Summary

We've covered a lot of ground in this chapter, much of it quite broad in focus. The aim of this chapter was to introduce the root concepts we'll be using in the rest of the book without this information cluttering up the details of what we'll aim to implement there.

At this point you know how the system manages authentication and authorization, and how that process is defined. You know how user profile data is stored on the system and how it is accessed. Lastly, you know about Mach ports and messaging, and that this is the means by which the Mach kernel performs most of its work. This last item forms the key of our first task: crash reporting.

Chapter 2

Crash Reporting

In this chapter we will look at the tasks involved in implementing a crash reporting tool of our own to replace Apple's default one when our own application crashes. We'll see how crashes can be caught and what information you can get out of them, then how you can show that information to the user and file it to your own webserver or email address.

Much of the information in this chapter came together with the aid of a message board post made by Tim Wood of the Omni Group a number of years ago[2]. For this I give many thanks.

2.1 When Things Go Wrong...

We've all seen the dialog shown in **Figure 2.1** at one time or another. This is what appears when an application crashes due to conditions like a memory access violation or a specific abort (caused by an uncaught exception for instance).

Crashing is a fact of life, and as developers we are if anything *more* likely to see these dialogs than our end users, as part of our job description is to track down and remedy such conditions. However, Apple has a very interesting tool for handling crashes which enables them to not only determine when an app has crashed but to gather information about its cause, the application's state, and to optionally package up and forward that information to Apple's bug reporting tool. This is a very powerful and useful piece of technology,

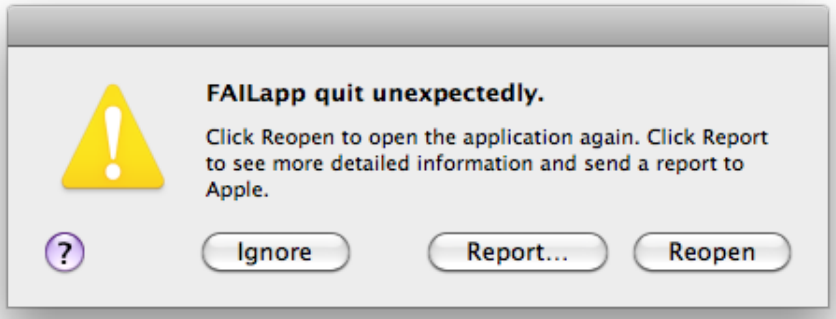

Figure 2.1: OMG FAIL!

and more than a few developers have wished that they had some way to hook into the system crash reporter dialog to get crashes from their own apps sent to them instead of Apple.

One third-party solution, from the clever folks at Unsanity, is the *Smart Crash Reports* haxie.[1] This uses the InputManager system to load their code into the CrashReporter application when it launches and makes some changes based on the application which crashed. If the crashed application has some specific keys in its `Info.plist` file then the Smart Crash Reports code will alter the resulting dialog to send the report to the developer of that application. Unfortunately, since it uses the InputManager system to load itself, it isn't compatible with OS X 10.6, which no longer loads InputManager plugins, thus cutting off the installation vector.

Another technique is to build a crash reporter infrastructure into your application directly, based on signal handlers; this approach is used by Landon Fuller's *PLCrashReporter* library.

With a little more work on our part, though, we can implement our own version of the Crash Reporter tool and bundle it with our own applications (and as system administrators, we could even install our own replacement for Apple's version and have the system use that instead). One example of this

[1] The term 'haxie', a play on the word 'hack', is used by the folks at Unsanity to refer to their pre-packaged system modifications.

approach is the Omni Group's *OmniCrashCatcher* application, built into all their Mac applications. We will develop something along the same lines in this chapter.

2.2 Catching A Crash

2.2.1 Signals and Exceptions

When an application does something untoward, various things happen at the system level. From a UNIX standpoint, a signal is sent to the process which caused the error; this is commonly `SIGINT`, `SIGBUS` or `SIGSEGV` for access violations and `SIGABRT` to manually crash the app in the event of an uncaught high-level exception. The default handlers for all these signals create a memory core dump of the application and terminate it, but by installing your own handler you can do whatever you please within the limits of operation inside a signal-handler.

The Mach kernel takes a more refined approach by using Mach messaging to send a low-level exception event to a handler for the target process. By default, OS X uses a Mach exception handler for all processes, and this causes CrashReporter to do its stuff. There's nothing preventing an application from overriding this however, and installing their own exception handler. Having done this, all exceptions are sent as high-level messages to a designated *exception port* created by the application. The only step is to create a thread to listen on that port. This can be a dedicated thread inside the application itself (although there are some nuances of which you should be wary in that case) or it can be a separate application. This latter approach is what we will use to build our crash catcher.

2.2.2 Catching Mach Exceptions

Mach exceptions are caught by creating a server which assigns itself as the destination for exception messages from a particular process (or all processes). The methods used by this service are shown in `exc_server.h`, which may be located at `<mach/exc_server.h` or in the same location within `Kernel.framework`, depending on your operating system version. On OS X 10.6, they are in the latter, but the content of the file hasn't changed, only its location.

The first step is to create a mach port to receive messages. For this we use the `mach_port_allocate()` function. When we allocate a port, we specify a simple *right* associated with that port. In this case, as we want to receive messages from the kernel, we specify `MACH_PORT_RIGHT_RECEIVE` at this point. This only gives half of the required rights however. We thus far have a port from which our process can read, but nothing has the right to *write* to it. To remedy this, we need to add another right to the port, which we do through the `mach_port_insert_right()` function. What we want is for any process attaching to our *receive-right* to be able to get a *send-right* which delivers messages to its message queue. Additionally, we want to ensure that each sender has its own send-port, that they're not shared amongst multiple processes (sharing is not a requirement, so this is the safest and most reliable option). The right we want to add in this case is a *make-send* right, which tells the kernel that it can generate new send-rights with accompanying send ports as they are requested. This is denoted by the `MACH_MSG_TYPE_MAKE_SEND` constant. The full allocation process can be seen in Listing 2.1.

Listing 2.1: Allocating an Exception Server Port

```
1  #include <mach/message.h>
2  #include <mach/mach_error.h>
3  #include <mach/task.h>
4  #include <mach/port.h>
5
6  kern_return_t kr = KERN_SUCCESS;
7  mach_port_t exc_catcher_port = MACH_PORT_NULL;
8
9  kr = mach_port_allocate(mach_task_self(), ↩
       MACH_PORT_RIGHT_RECEIVE, &exc_catcher_port);
10 if ( kr != KERN_SUCCESS )
11 {
12     fprintf(stderr, "mach_port_allocate(): %d (%s)\n", ↩
           kr, mach_error_string(kr));
13     return;
14 }
15
16 kr = mach_port_insert_right(mach_task_self(), ↩
       exc_catcher_port, exc_catcher_port, ↩
       MACH_MSG_TYPE_MAKE_SEND);
17 if ( kr != KERN_SUCCESS )
18 {
```

```
19      fprintf(stderr, "mach_port_insert_right(): %d (%s)\↩
            n", kr, mach_error_string(kr));
20      return;
21  }
```

Next we want to be able to asynchronously receive messages on this port. In the past, we would spawn a new thread which would loop around calls to `mach_msg()` or `mach_msg_overwrite()`, or we would use a MIG-generated `exc_server_routine()` function to do that for us. With a little CoreFoundation or Foundation help we could wrap the port and place it into a runloop. In C we might use the `kevent()` API to find out when a message arrives— but that would require another background thread and/or more glue to make it work asynchronously.

In OS X 10.6 we have a better way: Grand Central Dispatch. Using this API we can create a *dispatch source* which monitors a given Mach port for the arrival of new messages, and we can provide a *Block* of code for that source to run when a message arrives on the port. This source can then be attached to an *event queue* (specifically, the one for the application's main thread) and will enqueue the provided block there when messages arrive. This enables us to be purely asynchronous without the need for spawning extra threads or using additional APIs, and without the need for complex resource-sharing algorithms such as locks or semaphores. Additionally, the use of C Blocks, a form of lexical closure, means that we needn't worry about the details of keeping relevant contextual information around— the Blocks runtime will do all of that for us for heap and stack variables alike.

Listing 2.2 shows the process of setting up a dispatch source to handle messages on our exception port. Note that it handles messages by calling out to the MIG-generated `exc_server()` function to handle decode and dispatch of the message (yes, just about every Mach API is generated using MIG).

Listing 2.2: Creating a Dispatch Source for our Exception Port

```
1  dispatch_source_t exc_source = dispatch_source_create( ↩
       DISPATCH_SOURCE_TYPE_MACH_RECV, exc_catcher_port,↩
        0, dispatch_get_main_queue() );
2
3  // handle new messages when they arrive
4  dispatch_source_set_event_handler( exc_source, ^{
5      mach_msg_header_t *msg, *reply;
6      kern_return_t krc = KERN_SUCCESS;
```

```
7
8  #define MSG_SIZE 512
9      msg = alloca(MSG_SIZE);
10     reply = alloca(MSG_SIZE);
11
12     // read the incoming message
13     krc = mach_msg( msg, MACH_RCV_MSG, MSG_SIZE, ↩
           MSG_SIZE, exc_catcher_port, 0, MACH_PORT_NULL↩
            );
14     MACH_CHECK_ERROR(mach_msg, krc);
15
16     // demux and dispatch the message to the ↩
           appropriate handler
17     if ( exc_server(msg, reply) == false )
18     {
19         fprintf( stderr, "exc_server() hated the ↩
               message" );
20         return;     // returns from the block
21     }
22
23     // we've got the reply from the MIG handler ↩
           function
24     // now we need to send it back
25     (void) mach_msg( reply, MACH_SEND_MSG, reply->↩
           msgh_size, 0, msg->msgh_local_port, 0, ↩
           MACH_PORT_NULL );
26 });
```

We now know how we're going to receive exceptions sent by the kernel, but we haven't told the kernel that we want to receive them, or for whom. The first piece of information we need is the target's *task port*. This is the Mach equivalent of a process ID, and in fact you can map between the two easily. However, in order to gain access to another task's task port your application will need to either run as the root user or as a member of the `procmod` group using the `setgid` file flag. Once we have a task port, we can take a copy of the existing exception handler ports and other metadata before installing our own handler. The code in Listing 2.3 shows how this is done.

Listing 2.3: Setting Exception Ports

```
1  // define our handler data structure
2  #define HANDLER_COUNT 64
3  typedef struct _ExceptionPorts {
4      mach_msg_type_number_t  maskCount;
```

```
5       exception_mask_t        masks[HANDLER_COUNT];
6       exception_handler_t     handlers[HANDLER_COUNT];
7       exception_behavior_t    behaviors[HANDLER_COUNT];
8       thread_state_flavor_t   flavors[HANDLER_COUNT];
9   } ExceptionPorts;
10  static ExceptionPorts * gOldHandlerData = NULL;
11
12  ...
13
14  pid_t procID = ...;     // get target process ID
15  task_t task = MACH_PORT_NULL;
16  kern_return_t kr = KERN_SUCCESS;
17
18  kr = task_for_pid( mach_task_self(), procID, &task );
19  if ( kr != KERN_SUCCESS )
20  {
21      fprintf( stderr, "task_for_pid: %d (%s)\n", kr, ↩
            mach_error_string(kr) );
22      return;
23  }
24
25  gOldHandlerData = calloc(1, sizeof(gOldHandlerData));
26  gOldHandlerData->maskCount = HANDLER_COUNT;
27  kr = task_get_exception_ports( task, EXC_MASK_ALL, ↩
        gOldHandlerData->masks, gOldHandlerData->↩
        maskCount, gOldHandlerData->behaviors, ↩
        gOldHandlerData->flavors );
28  if ( kr != KERN_SUCCESS )
29  {
30      fprintf( stderr, "task_get_exception_ports: %d (%s)↩
            \n", kr, mach_error_string(kr) );
31      return;
32  }
33
34  // install new ports
35  kr = task_set_exception_ports( task, EXC_MASK_ALL & ~(↩
        EXC_MASK_MACH_SYSCALL|EXC_MASK_SYSCALL|↩
        EXC_MASK_RPC_ALERT), exc_catcher_port, ↩
        EXCEPTION_DEFAULT, THREAD_STATE_NONE );
36  if ( kr != KERN_SUCCESS )
37  {
38      fprintf( stderr, "task_set_exception_ports: %d (%s)↩
            \n", kr, mach_error_string(kr) );
39      return;
40  }
```

You can see above that we're handling all exception types except for Mach system calls, UNIX system calls, and RPC alerts (Mach messaging). These types of exceptions are used to implement basic kernel-to-user interface functionality, while the others are all (potentially at least) seen when an application crashes.

In addition to setting the exception ports, we need to revert back to the original ones when our application is done. Since we're using Grand Central Dispatch we can set a *cancel handler* on our exception event source. This way we can simply cancel the handler and everything will be reset for us using code which we define, seen here in Listing 2.4.

Listing 2.4: Resetting Exception Ports

```
1  dispatch_source_set_cancel_handler( exc_source, ^{
2      // reinstall the old exception ports
3      int i;
4      for (i = 0; i < gOldHandlerData->maskCount; i++)
5      {
6          if ( gOldHandlerData->handlers[i] == ↩
                MACH_PORT_NULL )
7              break;
8          task_set_exception_ports( task, gOldHandlerData↩
                ->masks[i], gOldHandlerData->handlers[i],↩
                 gOldHandlerData->behaviors[i], ↩
                gOldHandlerData->flavors[i] );
9      }
10
11     // destroy our exception handler port
12     mach_port_destroy( mach_task_self(), ↩
             exc_catcher_port );
13 });
```

When releasing our receive port, we use `mach_port_destroy()` to ensure that the port is destroyed. The similar `mach_port_deallocate()` function will decrement the port's reference count, causing it to be released only once that reference count reaches zero. When writing the server-side of these things, I prefer to destroy the server port explicitly rather than risk leaving it open somewhere.

Note also that we use a *for loop* to install the old handlers. This is because there may be many different ports used to handle the different types of message, and these ports might belong to many different processes. For

instance, a memory paging system might install a handler for `bad access` exceptions, handling them by paging in storage from disk.

2.2.3 Task Death

At this point, we want to think a little about our target application. What should happen if that task exits cleanly? Well, in this case we want to find out about it and stop running. In the UNIX world we might use `wait()` or `waitpid()` to accomplish this, but again these are synchronous calls. A better means would be to wait for a *port death* notification on our target's task port, again using a dispatch source as an asynchronous event processor.

Anyone with rights to a Mach port can request a notification of one of several exceptional states, found in `<mach/notify.h>`:

1. **Port Deleted**
   A send or send-once right was deleted. This notification is sent to the owner of the receiving port.
2. **Port Destroyed**
   A receive port was destroyed. The notify message retains the receive right, however, preventing it from being destroyed until the notification handler calls `mach_port_destroy()` or `mach_port_deallocate()`.
3. **No Senders**
   A receive port's last sender was deallocated. At this point an on-demand Mach server can safely shut down. This message is sent to the owner of the receiving port.
4. **Send Once**
   A port created via the make-send-once right was deallocated.
5. **Dead Name**
   A send (or send-once) right died, leaving a dead name in the current namespace. A *dead name* is similar in concept to a dangling pointer in C, except that it won't cause a crash when you attempt to use it, just an error.

In our situation, we can use the *dead name* notification to find out when a task port is deleted. As a task port is only deleted when the corresponding task is destroyed, this notification will tell us when our target task has terminated. Listing 2.5 shows how this is done.

Listing 2.5: Handling Task Death Notifications

```
1  mach_port_t death_port = MACH_PORT_NULL;
2  mach_port_t old_port = MACH_PORT_NULL;
3  kr = mach_port_allocate( mach_task_self(), ↩
       MACH_PORT_RIGHT_RECEIVE, &death_port );
4  kr = mach_port_request_notification( mach_task_self(), ↩
       [container task], MACH_NOTIFY_DEAD_NAME, 0, ↩
       death_port, MACH_MSG_TYPE_MAKE_SEND_ONCE, &↩
       old_port );
5
6  dispatch_source_t death_source;
7  death_source =dispatch_source_create( ↩
       DISPATCH_SOURCE_TYPE_MACH_RECV, death_port, 0, ↩
       dispatch_get_main_queue() );
8
9  dispatch_source_set_cancel_handler( death_source, ^{
10     // when cancelled, reset the old notification port
11     // (this may be MACH_PORT_NULL)
12     mach_port_request_notification( mach_task_self(), ↩
           task, MACH_NOTIFY_DEAD_NAME, 0, old_port, 0, ↩
           NULL );
13
14     // the source owns our port, so we destroy it here
15     mach_port_destroy( mach_task_self(), death_port );
16 });
17
18 dispatch_source_set_event_handler( death_source, ^{
19     mach_msg_header_t * msg;
20     msg = alloca(MSG_SIZE);
21
22     // consume the message (we know what it says)
23     (void) mach_msg( msg, MACH_RCV_MSG, MSG_SIZE, ↩
           MSG_SIZE, death_port, 0, MACH_PORT_NULL );
24
25     // the task has gone-- log the crash report
26     // we only do this once the app has gone, in case
27     // a logged exception isn't fatal
28     backtrace_log();
29
30     // the task we're watching is gone: quit now
31     dispatch_source_cancel( exc_source );
32     dispatch_source_cancel( death_source );
33 });
```

One important detail here is the call to `backtrace_log()` on line 27. We

actually record a backtrace on every exception, even though that backtrace might not result in the application's termination. As a result, we want to store that trace until such time as we know the application has terminated. The way we do that here (although this is over-simplified in the extreme) is to only log the backtrace information when the application terminates. If there is no backtrace, then we simply exit. Otherwise, we generate a log file. We will see this in action in Section 2.3.2.

2.2.4 Running the application

We're now handling exceptions and task termination asynchronously using blocks. We've not had to implement any other functions (yet), although we'll have to implement the exception handler routines in a moment. Right now however, we've only one small task to do: fire up our sources and run the dispatch processing loop. This simple task, the last item in our `main()` function, is shown in Listing 2.6.

Listing 2.6: Running the Dispatch Loop

```
1  // dispatch sources are created in a suspended state,
2  // so we must resume them now
3  dispatch_resume( exc_source );
4  dispatch_resume( death_source );
5
6  // run the dispatch loop
7  dispatch_main();
8
9  // once that returns, we're all done
10 return ( 0 );
```

2.3 Exception Handling

2.3.1 Implementing Mach Exception Handlers

There are three Mach exception routines which we need to implement, each serving a slightly different purpose:

- **`catch_exception_raise()`**
  This is the simplest function. Its parameters specify the exception which occurred along with the task and thread on which it happened.

- `catch_exception_raise_state()`
  This version allows the caller to alter the state of a thread. It passes in the current thread state, and provides storage for a new state to be applied. It does not, however, provide the ports for either the task or the thread on which the exception occurred.

- `catch_exception_raise_state_identity()`
  This is the best of both worlds. It provides the task and thread ports for the triggering thread, and it also allows the handler to directly inspect and update the thread's state.

For our purposes, we will have to implement all three methods, along with a method to forward the exceptions to their original handler(s). We will look at this method first, as it shows the methodology behind processing exceptions to a certain degree.

We'll call our method `forward_exception()`, and you can see it in full in Listing 2.7.

Listing 2.7: Implementing `forward_exception()`

```
1  kern_return_t
2  forward_exception
3  (
4      thread_t thread,
5      mach_port_t task,
6      exception_type_t exception,
7      exception_data_t code,
8      mach_msg_type_number_t codeCount,
9      int *flavor,
10     thread_state_t old_state,
11     mach_msg_type_number_t old_stateCnt,
12     thread_state_t new_state,
13     mach_msg_type_number_t *new_stateCnt
14 )
15 {
16     kern_return_t kr;
17     unsigned int portIndex;
18
19     mach_port_t port;
20     exception_behavior_t behaviour;
21     int thread_flavor;
22
```

```
23      thread_state_data_t thread_state;
24      mach_msg_type_number_t thread_state_count;
25
26      for ( portIndex = 0; portIndex < gOldHandlerData->↩
              maskCount; portIndex++ )
27      {
28          if ( gOldHandlerData->masks[portIndex] & (1 << ↩
                  exception) )
29          {
30              // This handler wants the exception
31              break;
32          }
33      }
34
35      if ( portIndex >= gOldHandlerData->maskCount )
36      {
37          fprintf( stderr, "No handler for exception type↩
                    %d. Not forwarding.\n" );
38          return ( KERN_FAILURE );
39      }
40
41      port = gOldHandlerData->handlers[portIndex];
42      behaviour = gOldHandlerData->behaviors[portIndex];
43      flavor = gOldHandlerData->flavors[portIndex];
44
45      fprintf( stderr, "Forwarding exception, port = %#x,↩
                behaviour = %d, flavor = %d\n", port, ↩
              behaviour, flavor );
46
47      if ( (behaviour != EXCEPTION_DEFAULT) &&
48           (old_state == NULL) )
49      {
50          thread_state_count = THREAD_STATE_MAX;
51          kr = thread_get_state( thread, &thread_flavor, ↩
                  thread_state, &thread_state_count );
52          MACH_CHECK_ERROR_RET(thread_get_state, kr);
53
54          flavor = &thread_flavor;
55          old_state = thread_state;
56          old_stateCnt = thread_state_count;
57
58          new_state = thread_state;
59          new_stateCnt = &thread_state_count;
60      }
61
```

```
62      switch ( behaviour )
63      {
64          case EXCEPTION_DEFAULT:
65              fprintf( stderr, "Forwarding to ↩
                    exception_raise\n" );
66              kr = exception_raise( port, thread, task, ↩
                    exception, code, codeCount );
67              MACH_CHECK_ERROR_RET(exception_raise, kr);
68              break;
69
70          case EXCEPTION_STATE:
71              fprintf( stderr, "Forwarding to ↩
                    exception_raise_state\n" );
72              kr = exception_raise_state( port, exception↩
                    , code, codeCount, flavor, old_state,↩
                     old_stateCnt, new_state, ↩
                    new_stateCnt );
73              MACH_CHECK_ERROR_RET(exception_raise_state,↩
                     kr);
74              break;
75
76          case EXCEPTION_STATE_IDENTITY:
77              fprintf( stderr, "Forwarding to ↩
                    exception_raise_state_identity\n" );
78              kr = exception_raise_state_identity( port, ↩
                    thread, task, exception, code, ↩
                    codeCount, flavor, old_state, ↩
                    old_stateCnt, new_state, new_stateCnt↩
                     );
79              MACH_CHECK_ERROR_RET(↩
                    exception_raise_state_identity, kr);
80              break;
81
82          default:
83              fprintf( stderr, "forward_exception: ↩
                    unknown beaviour %d\n", behaviour );
84              break;
85      }
86
87      if ( behaviour != EXCEPTION_DEFAULT )
88      {
89          kr = thread_set_state( thread, *flavor, ↩
                 new_state, *new_stateCnt );
90          MACH_CHECK_ERROR_RET(thread_set_state, kr);
91      }
```

```
92
93      return ( KERN_SUCCESS );
94  }
```

There's a lot of code in here, so let's cover it piecemeal.

- **Lines 4–13:**
  We take the largest set of parameters possible, but we don't require that they all be valid. We can't figure out the source task or thread without being given those details, but we can get a thread's state ourselves if we have been passed a thread port to use.
- **Lines 26–33:**
  Here we look for the first handler which is interested in the exception we're forwarding. Once we find one, we exit the loop— only the first will get to process the exception.
- **Lines 41–33:**
  Get the exception details from the the old handler's data.
- **Lines 47–60:**
  If the old handler's behaviour indicates that it wants thread state information but we've not been passed any, fetch it using `thread_get_state()`. Assign the results to the parameter variables used by the following code.
- **Lines 62–85:**
  Based on the exception behaviour requested by the old handler, call the appropriate variant of `exception_raise()`, passing along the relevant parameters. The first argument to this function is the old handler's port, so it gets sent directly to that handler.
- **Lines 87–91:**
  If the old handler requested thread state information, then it has also given us some back. Use that to set the thread's state via a call to `thread_set_state()`.

This function should be called at the end of all our exception handlers. We only want to grab some details of the crashed application's state, we're not interested in trying to handle the exception and prevent the crash in any way.

The exception handlers themselves are reasonably straightforward. We want to get backtraces for every thread in the crashed task, so we only need the task port. This is provided for us in every instance with the exception of the `EXCEPTION_STATE` behaviour. However, we can work around this by keeping a copy of our target task port in a global variable to be referenced in this method. This leaves us with the rather simple declarations shown in Listing 2.8.

Listing 2.8: Exception Handler Implementation

```
1  kern_return_t
2  catch_exception_raise
3  (
4      mach_port_t exception_port,
5      thread_t thread,
6      mach_port_t task,
7      exception_type_t exception,
8      exception_data_t code,
9      mach_msg_type_number_t codeCnt
10 )
11 {
12     kern_return_t kr;
13     backtrace_task( task, thread, code, codeCnt );
14
15     kr = forward_exception( thread, task, exception, ↩
           code, codeCnt, NULL, NULL, 0, NULL, 0 );
16     return ( kr );
17 }
18
19 kern_return_t
20 catch_exception_raise_state
21 (
22     mach_port_t exception_port,
23     exception_type_t exception,
24     exception_data_t code,
25     mach_msg_type_number_t codeCnt,
26     int *flavor,
27     thread_state_t old_state,
28     mach_msg_type_number_t old_stateCnt,
29     thread_state_t new_state,
30     mach_msg_type_number_t *new_stateCnt
31 )
32 {
33     kern_return_t kr = KERN_SUCCESS;
```

```
34      backtrace_task( gTargetTask, MACH_PORT_NULL, code, ↩
              codeCnt );
35
36      kr = forward_exception( MACH_PORT_NULL, ↩
              MACH_PORT_NULL, exception, code, codeCnt, ↩
              flavor, old_state, old_stateCnt, new_state, ↩
              new_stateCnt );
37      return ( kr );
38  }
39
40  kern_return_t
41  catch_exception_raise_state_identity
42  (
43      mach_port_t exception_port,
44      mach_port_t thread,
45      mach_port_t task,
46      exception_type_t exception,
47      exception_data_t code,
48      mach_msg_type_number_t codeCnt,
49      int *flavor,
50      thread_state_t old_state,
51      mach_msg_type_number_t old_stateCnt,
52      thread_state_t new_state,
53      mach_msg_type_number_t *new_stateCnt
54  )
55  {
56      kern_return_t kr;
57      backtrace_task( task, thread, code, codeCnt );
58
59      kr = forward_exception( thread, task, exception, ↩
              code, codeCnt, flavor, old_state, ↩
              old_stateCnt, new_state, new_stateCnt );
60      return ( kr );
61  }
```

All of the items simply call through to `backtrace_task()` and then forward the exception on to its original handlers.

2.3.2 Backtracing

Now we reach the last big piece of the puzzle: backtracing. As of OS X 10.6, obtaining a nicely-printed backtrace of your own stack has become

much easier through the opening of the previously-private `backtrace()` and `backtrace_symbols()` API (see `<execinfo.h>` for more information).

These APIs only look at the stack of the calling thread however. This means that they're not much use in our situation, where we need to look at the stack of not only another thread, but another process in a foreign address space. Once again, Mach has our backs covered. In `<mach/vm_map.h>`, `<mach/vm_param.h>` and similar, we can see the API for accessing the virtual memory space of another task. All we need is the task port for the foreign process and we can read and write there to our hearts' content. This means that we could, for example, take the information from *More Backtrace* and update it for x86_64 processor support. Alternatively you might look at the *BOINC* project, which has an updated version of the same, albeit under the slightly more restrictive LGPL license.

However, in the interest of brevity, I shall do neither here, but rather will link to an OS X private framework— `Symbolication.framework`. This is the same framework which provides the official *ReportCrash* application's backtracing support, and it has a nice Objective-C API which you can see through judicious use of the *class-dump* tool[2]. Our use of the framework is quite limited (we use only five classes and one method on each of them) so I'll show the entire backtrace-generating routine first, in Listing 2.9, and describe what it's doing afterward.

Listing 2.9: A Simple Remote Backtracer

```
1  if ( gBacktraceLog == nil )
2      gBacktraceLog = [NSMutableString new];
3  else
4      [gBacktraceLog setString: @""];
5
6  VMUSymbolicator * symbolicator = [VMUSymbolicator ↩
       symbolicatorForTask: task];
7  NSArray * samples = [VMUSampler sampleAllThreadsOfTask:↩
        task withSymbolicator: symbolicator];
8  NSUInteger i = 0;
9  for ( VMUBacktrace * backtrace in samples )
10 {
```

[2]You might also want to look at the *class-dump-z* project which, while it doesn't support some things like 64-bit binaries, has a number of niceties missing in the vanilla *class-dump* implementation. The project is located at http://code.google.com/p/networkpx/wiki/class_dump_z

```
11      [gBacktraceLog appendFormat: @"Thread %d (%#x)", i,↩
              [backtrace thread]];
12      if ( [backtrace thread] == exc_thread )
13          [gBacktraceLog appendString: @" Crashed"];
14      [gBacktraceLog appendString: @":\n"];
15
16      pointer_t * trace = [backtrace backtrace];
17      for ( int j = 0; j < [backtrace backtraceLength]; j↩
              ++ )
18      {
19          VMUSymbol * symbol = [symbolicator ↩
                  symbolForAddress: trace[j]];
20          VMUSymbolOwner * owner = [symbolicator ↩
                  symbolOwnerForAddress: trace[j]];
21          [gBacktraceLog appendFormat: @"%d\t%-30s %p : %↩
                  @\n", j, [[owner name] UTF8String], trace↩
                  [j], [symbol name]];
22      }
23
24      // empty line between items
25      [gBacktraceLog appendString: @"\n"];
26
27      i++;
28  }
```

Let's look at that we're doing here:

- **Lines 1–4:**
  As discussed above, we will keep a backtrace log around until the target application terminates, since an exception might not lead to an outright crash. This is where we manage that log, in an `NSMutableString` instance.

- **Line 6:**
  The `VMUSymbolicator` class knows about obtaining symbolic information from a process's direct and indirect symbol tables. We create one up front to save on the potentially costly cross-process virtual memory copying and caching which this class uses to do its work.

- **Line 7:**
  The `VMUSampler` class generates the backtraces themselves. In this case we're using a simple one-shot method to get back an `NSArray` containing one `VMUBacktrace` for each thread in the target process.

- **Lines 9–28:**
  Here we loop through the results of backtracing the thread, with each frame being output like so:
  - **Lines 11–14:**
    First we append the details of the thread itself— its index and the Mach thread handle (port). If this thread is the same as the one which caused the exception, then we note that in the output as well.
  - **Line 16:**
    The `VMUBacktrace` instance has a `-backtrace` accessor which returns an array of pointers. These are the addresses of each stack frame in the backtrace.
  - **Lines 19–20:**
    We obtain a `VMUSymbol` describing the symbol at the frame address, and we get an object representing the owner of that symbol, i.e. the library from which it comes.
  - **Line 21:**
    We build the output line with the frame index followed by the library name padded to 30 characters with spaces, then the frame pointer and the symbol name.

An example of the output generated by this method can be seen in Listing 2.10.

Listing 2.10: Sample Backtracer Output

```
Thread 7 (0x3f03):
0   libSystem.B.dylib               0x0000008885efca : ↩
      __semwait_signal
1   libSystem.B.dylib               0x00000088862de1 : ↩
      _pthread_cond_wait
2   JavaScriptCore                  0x00000082f8d1a0 : ↩
      WTF::ThreadCondition::timedWait(WTF::Mutex&, ↩
      double)
3   WebCore                         0x00000080bd4dd1 : ↩
      WebCore::LocalStorageThread::threadEntryPoint()
4   libSystem.B.dylib               0x0000008885d536 : ↩
      _pthread_start
5   libSystem.B.dylib               0x0000008885d3e9 : ↩
      thread_start
```

2.4 Summary

In this chapter we have seen how the Mach kernel dispatches hardware and software exceptions to other processes through the use of Mach ports, and how those processes can catch and make use of them. We've covered the necessary details of the mach exception server setup and how to create a dedicated server to catch exceptions for a single task. We've also looked at a simple way of obtaining thread backtraces to create a crash log in our own format— from here it's a simple step to present some user interface similar to Apple's own crash reporting tool, offering the user the chance to email you the crash log with any other details they can provide.

You can view the complete source code for the examples presented in this chapter, along with those from other chapters, at http://github.com/alanQuatermain/secret-sauce.

Chapter 3

Managed Client

In this chapter we will put aside code for a brief moment and look at the mechanics of the Managed Client system on OS X, also known as *Parental Control*. We'll find out where this data is stored, in what formats, and how it is used by applications. To do this we will have to look at the OS X directory service and the preferences system, including a peek into the code for the open-source components of CoreFoundation.

3.1 MCX

3.1.1 What is MCX?

MCX stands for **M**anaged **C**lient for OS **X**[3] and was originally designed for use on Mac OS X Server version 10.2 (Jaguar). It is formally defined as a subset of Open Directory, Apple's directory service, which we looked at briefly in Section 1.2, and is used to implement access policies for client systems. These access policies are used to define access rights to many different items, including individual applications or dock widgets, websites, email and chat recipients, and more. The MCX records themselves are designed to be set on any number of LDAP records, whether for users, groups, computers, or computer groups. They are then cached by client machines and compiled into a single authoritative document which applies to the currently-logged-in user of the machine. In the parental controls scenario, the items are applied only to individual user records, and not as many options are provided, but otherwise their handling is identical.

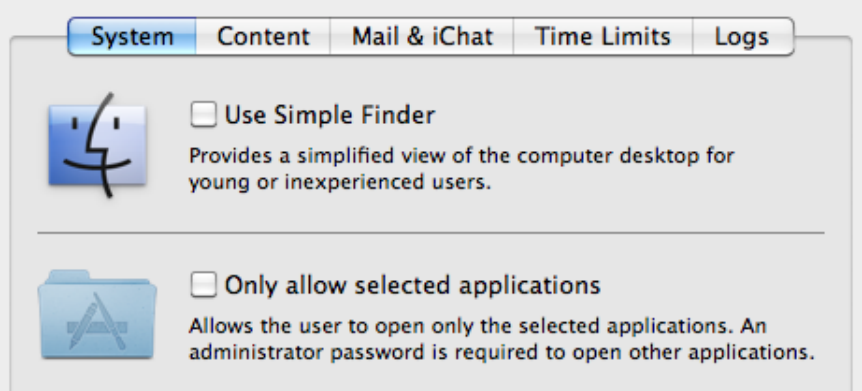

Figure 3.1: Parental Controls on Mac OS X

From an Authentication vs. Authorization standpoint (see Section 1.3.1), the Directory Service itself (whether it's an Apple Open Directory, Microsoft Active Directory, Novell eDirectory, or LDAP) first provides user *authentication*. The MCX settings then define that user's *authorization* with regards to the computer that they are using. As we will see in Section 3.2.2, the MCX settings are generally stored as a single data block inside a client directory type. OS X Server usually requires that this be an Open Directory server.

3.1.2 Server MCX vs. Parental Controls

There are a few differences between the types of MCX settings you can manage using OS X Server and the Parental Controls preferences on your Mac workstation, the latter being a subset of the former. One major example is that the server allows you to differentiate between *set-once* items and *forced* items, while Parental Controls only creates forced ones. Set-once items are designed to make a change which the user isn't required to keep, but which is recommended. An example would be to add a link to a university's student support website on a user's dock— it makes it easily available to the user, but they are free to remove it if they don't need that link stored there. A forced item is a permanent restriction. For instance, prohibiting a user from burning CDs or DVDs (or both) is a commonly forced setting.

Another difference revolves around the use of print services. OS X Server provides management facilities for shared print queues exported via Open Directory. Server MCX settings allow an administrator to place restrictions

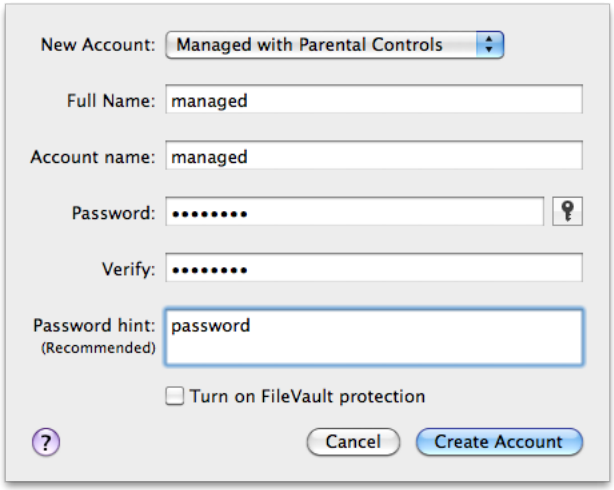

Figure 3.2: Creating a Managed User Account

on the print jobs that certain users (or groups, computers, or labs) can send to each queue. Parental Controls doesn't include that restriction, primarily because such things are usually the sole concern of computer networks with shared resources.

For the remainder of this section we will be working on an OS X workstation, and thus will only see the items which can be modified using the Parental Controls preference pane shown in Figure 3.1.

3.2 MCX Settings

3.2.1 Creating a Test Account

Our first task is to create a new user account with which to test the MCX implementation. This begins with a trip to the System Preferences app and the Accounts preference pane. Unlock the pane and add a new account. On the sheet which appears (cf. Figure 3.2), pull down the *New Account* popup and select 'Managed with Parental Controls'. Fill in the rest of the details however you see fit, but make a note of the password.

Once this is done, select the new account in the source list to the left and click the 'Open Parental Controls' button to see the Parental Controls preference

pane. Here you can activate any options you choose; we will show the different forms of output data for each option later in the chapter, so feel free to select anything you like.

3.2.2 Open Directory

This is where our dive into the `dscl` command-line tool in Section 1.2.2 comes in use. We've created the user account, and we know that MCX is a subset of the Open Directory schema, so the first place we will look for that data is inside Open Directory. Specifically, we will look at the record for the user we've just created.

Open a Terminal window and enter the commands shown in Listing 3.1.

Listing 3.1: Locating the User Record

```
Jims-iMac ~ $ dscl localhost
 > cd Local/Default/Users/
/Local/Default/Users > read managed
...
```

There's a lot of output now, isn't there? The most interesting part, however, is the new *MCXSettings* attribute, which appears to be a fairly standard property list, as excerpted in Listing 3.2.

Listing 3.2: The *MCXSettings* attribute

```
<?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>mcx_application_data</key>
    <dict>
        <key>com.apple.Dictionary</key>
        <dict>
            <key>Forced</key>
            <array>
                <dict>
                    <key>mcx_data_timestamp</key>
                    <date>2010-12-07T21:24:08Z</date>
                    <key>mcx_preference_settings</key>
```

```
15                      <dict>
16                          <key>parentalControl</key>
17                          <true/>
18                      </dict>
19                  </dict>
20              </array>
21          </dict>
22          ...
23      </dict>
24  </dict>
25  </plist>
```

If you alter your settings in the Parental Controls preference pane and read this record again, you will see the updates reflected inside this property list. Everything about MCX happens here.[1]

3.2.3 Where The Magic Happens

We've seen where the data is kept. The next part of the puzzle is to locate it at runtime. Let's look at some of the keys in one of the smaller items and see what we might deduce:

- **`mcx_application_data`**
  This surrounds every we've seen so far. From its name, it seems safe to deduce that this part of the settings refers specifically to *application-related* settings. This also leads us to suspect that there may be another form of setting contained in this dictionary, perhaps something for use by OS X Server.
  - **`com.apple.Dictionary`**
    This looks like a bundle identifier, and after further investigation it is indeed the bundle identifier for the OS X Dictionary application. It seems most likely that this dictionary contains settings specific to that application.
    * **`Forced`**
      This seems to fit with the description of *set-once* vs. *forced* settings as defined in Section 3.1.2. It seems safe to deduce,

[1] On OS X Server there is also an attribute called *mcx-flags*. This is only used on exported directory services, and assists clients in merging multiple MCXSettings items, so we won't cover it here.

therefore, that this contains any settings which are to be considered immutable by the user.

- `mcx_data_timestamp`
  This looks like the date and time at which the item was written to the directory record, in RFC-3339 format. Changing values and re-checking the contents of the property list confirms this assumption. Given that the MCXSettings attribute is designed to be merged together with similar values from other directory records, this is likely used to help determine a precedence order during that merge.
- `mcx_preference_settings`
  This looks very much like the root of a regular preference property list, such as that in which user settings are commonly stored. Looking at different items within the MCXSettings attribute reinforces this conclusion.

What we can deduce, then, is that each item in the `mcx_application_data` dictionary contains data keyed for a specific application. Then there are both forced and set-once lists of preference settings for that application contained within there. The next question, however, is where this information comes into play? We know that the data is designed to be merged together from multiple such records, but we don't know what happens to it.

The next step, then, is to think laterally. This is property-list preference data, and preferences in this format are (ultimately) handled through CoreFoundation. Luckily for us, CoreFoundation is largely open-sourced[2], so we can take the advice of many UNIX sages before us[3] and go straight to the source.

Our first stop is `CFApplicationPreferences.c`, since this implements the standard search path used to lookup preference values for an application. A search of this file for the term *MCX* turns up nothing, but when we look for the word *managed* we find the interesting comment excerpted in Listing 3.3.

[2] A lot of Apple's OS X code can be found at `http://www.opensource.apple.com`, including this. Its project-name is `CF-<version>`.

[3] "Use the source, Luke."

Listing 3.3: Managed Preferences?

```
1  /* Here is how the domains end up in priority order in ↩
       a search list.  Only a subset of these are setup ↩
       by default.
2  argument domain
3  this app, this user, managed
4  this app, any user, managed
5      this app, this user, this host
6      this app, this user, any host (AppDomain)
7  suiteN, this user, this host
8  suiteN, this user, any host
9      ...
10 */
```

You'll note that on lines 3–4 of the listing we see the word 'managed'. The following two entries on lines 5–6 are quite similar, and will appear familiar to anyone who has used the CFPreferences API, where a preference can be obtained by specifying an app-user-host tuple.

So we appear to have found a reference to a special domain which is overridden only by values passed on the command-line (well *there's* a loophole), and which overrides all other application preferences. We now need to find its details. Sadly, the managed domain is only mentioned in this one location within this file, and it isn't added to the search list by default. So what now?

Well unfortunately, the CoreFoundation source code isn't *all* open-sourced, and the details of the managed preference data is in the closed-source portion. We still have some more tricks up our sleeve though.

Armed with the knowledge that it's referred to as the 'managed' preference domain, we shall pull out a copy of `strings`. This is a useful command-line tool shipped with the standard Xcode development kit which will print out everything in the strings table of an application binary. Let's start by looking for 'managed'— see the results in Listing 3.4.

Listing 3.4: Searching for 'managed' in CoreFoundation

```
1  [jim@Quatermain:~]% strings /System/Library/Frameworks/↩
       CoreFoundation.framework/CoreFoundation| grep ↩
       managed
2  managed/%@/%@
3  managed/
4  [jim@Quatermain:~]%
```

That doesn't look entirely helpful. Perhaps capitalizing the word will help?

Listing 3.5: Searching for 'Managed' in CoreFoundation

```
[jim@Quatermain:~]% strings /System/Library/Frameworks/↩
      CoreFoundation.framework/CoreFoundation| grep ↩
      Managed
/Library/Managed Preferences
/Library/Managed Preferences/
/Library/Managed Preferences/%@/
/Library/Managed Preferences/%@
CFXPreferencesManagedSource
[jim@Quatermain:~]%
```

That looks more like it. Not only can we see a symbol called `CFXPreferencesManagedSource` which we might save for later, but we can see '/Library/Managed Preferences' in there too, which looks like the jackpot. Interestingly, we can also see format strings which would seem to indicate that this folder would contain *subfolders* rather than files. This looks like an interesting nuance.

Let's take a look at the output then. We'll switch on everything we can find in Parental Controls for text account, then we'll log in as that user to see what we can find. The results are in Listing 3.6.

Listing 3.6: Inside the Managed Preferences Folder

```
[jim@Quatermain:~]% cd /Library/Managed\ Preferences
[jim@Quatermain:..anaged Preferences]% ls
test
[jim@Quatermain:..anaged Preferences]% cd test
[jim@Quatermain:..eferences/test]% ls
com.apple.Dictionary.plist
com.apple.DiscRecording.plist
com.apple.applicationaccess.new.plist
com.apple.applicationaccess.plist
com.apple.dashboard.plist
com.apple.dock.plist
com.apple.familycontrols.contentfilter.plist
com.apple.familycontrols.logging.plist
com.apple.familycontrols.timelimits.plist
com.apple.finder.plist
com.apple.frameworks.diskimages.plist
com.apple.iChat.AIM.plist
com.apple.iChat.Jabber.plist
```

```
com.apple.iChatAgent.plist
com.apple.mail.plist
com.apple.mcxprinting.plist
com.apple.parentalcontrols.cache.plist
com.apple.screencapture.plist
com.apple.systempreferences.plist
com.apple.systemuiserver.plist
complete.plist
[jim@Quatermain:..eferences/test]%
```

So there we have it— the end result is a set of perfectly normal preference files. These files are owned and can only be modified by the root user, since they are potentially used to implement security policies.

3.3 MCX Preference Implementation

The contents of the `com.apple.Dictionary.plist` output by MCX is shown in Listing 3.7. Comparing it to the values in Listing 3.2 above, we can see that the output file apparently consists of the contents of the `mcx_preference_settings` dictionary.

Listing 3.7: Dictionary Application Managed Preferences

```
<?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>parentalControl</key>
    <true/>
</dict>
</plist>
```

The Dictionary application is fairly basic however. The really interesting stuff (from a third-party perspective) includes the settings for disc burning, content filters, and application access restrictions. Let's take a look at each of those in turn.

3.3.1 Disc Burning

Disc burning settings are spread across a number of applications. Firstly, `com.apple.DiscRecording.plist` specified a single key-value pair: *BurnSupport=off*. This will disable disc burning via the DiscRecording framework used by third-party applications. Additionally, the finder is given a setting of *ProhibitBurn=true* to disable its disc-burning options.

The last change is applied to the SystemUIServer application, which is responsible for determining the actions undertaken when blank media is inserted. In this case, the preferences contain a dictionary item with two sub-items, shown in Listing 3.8. There we can see two sub-items, *blankcd* and *blankdvd*, each of which has the same array of actions to take when these types of media are mounted by the system: *eject* the media, then *alert* the user.

Listing 3.8: SystemUIServer Settings to Disable Disc Burning

```
<?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>mount-controls</key>
    <dict>
        <key>blankcd</key>
        <array>
            <string>eject</string>
            <string>alert</string>
        </array>
        <key>blankdvd</key>
        <array>
            <string>eject</string>
            <string>alert</string>
        </array>
    </dict>
</dict>
</plist>
```

3.3.2 Web, Mail, and iChat Content Filters

The Parental Controls system allows for the specification of whitelists of web content, email recipients, and iChat message recipients. The specification of each of these can be found in the settings for each application.

The first and simplest is the web content filter, which can be seen in `com.apple.familycontrols.contentfilter.plist`. Browser vendors are suggested to check this preference domain before showing URLs, but for URLs originating outside the browser, OS X Launch Services will use the contents of this preference domain automatically. In our case, we've decided to stick with the built-in content filters rather than specify our own white list, and this is specified using the settings in Listing 3.9. The *restrictWeb* setting tells the system to use its built-in filtering rules, while the *useContentFilter* setting simply turns the filter on or off.

Listing 3.9: Web Content Filters

```
<?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>restrictWeb</key>
    <true/>
    <key>useContentFilter</key>
    <true/>
</dict>
</plist>
```

Our next subject, the Mail app, allows the specification of an email recipient whitelist and a parental permission-request email address to which Mail will automatically send emails when requested. This address is also used in the iChat whitelist discussed below. The settings are enabled through the *parentalControl* key, and the parental email is actually an array under the *parentalEmails* key. The actual email recipient whitelist is an array of dictionaries, as seen in Listing 3.10.

Listing 3.10: Mail MCX Settings

```
<?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>parentEmails</key>
    <array>
        <string>xxxxxxxx@mac.com</string>
    </array>
    <key>parentalControl</key>
    <true/>
    <key>whiteList</key>
    <array>
        <dict>
            <key>email</key>
            <string>xxxxxxx@apple.com</string>
        </dict>
    </array>
</dict>
</plist>
```

iChat's content filters are spread across a few different preference files, one for each of the Jabber and AIM protocols, and another for the iChatAgent background application. This latter is responsible for managing the iChat service connections and message delivery. The whitelist for each protocol is similar in structure to the Mail app's version, albeit with the *email* key replaced with *screenName*. The iChatAgent settings are used to enable parental controls and enable logging, as seen in Listing 3.11.

Listing 3.11: iChatAgent MCX Settings

```
<?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>Setting.parentalControls</key>
    <true/>
    <key>Setting.parentalControls.forceChatLogging</key↩
         >
    <true/>
</dict>
</plist>
```

3.3.3 Application Access

Application access settings are specified in `com.apple.applicationaccess.new.plist`. This contains a single item, *familyControlsEnabled*, to turn the filter on and off, and an array under the key *whiteList*. Each application is specified by the dictionary shown in Listing 3.12. Here we can see the application's bundle identifier, display name, and its path, but when this is generated on OS X Server it will also include a code-signing credential encoded as a data object. This provides both greater security, since the authorization will fail if an app has been tampered with, and will also enable the right to follow the application around if its path changes.

Listing 3.12: Application Access Whitelist Items

```
<dict>
    <key>bundleID</key>
    <string>com.apple.AddressBook</string>
    <key>displayName</key>
    <string>Address Book</string>
    <key>path</key>
    <string>/Applications/Address Book.app</string>
</dict>
<dict>
    <key>bundleID</key>
    <string>com.apple.Dictionary</string>
    <key>displayName</key>
    <string>Dictionary</string>
    <key>path</key>
    <string>/Applications/Dictionary.app</string>
</dict>
```

3.4 Summary

The knowledge of the formats used for handling managed preferences means that it is relatively easy to implement your own user interface around the creation of these settings. Looking at the output of OS X Server's MCX settings can show you how to augment OS X's Parental Controls system with other items available only from Mac OS X Server. Additionally, it should be noted that MCX also supports the setting of arbitrary preference values, so you can manage preferences for any application you choose.

If you're a little more daring, however, there is also the possibility of using a spawned process running as the root user to tweak the settings within the Managed Preferences folder at login. This has certain risks due to potential race conditions, but you could make use of the FSEvents API to track the creation of those files to update them as soon as possible after their creation, which might mitigate the risk somewhat.

Chapter 4

Complex Authorization

Coming Soon...

Chapter 5

Time Machine

This chapter will look in detail at an Apple Private API. Our target here will be the Time Machine user interface used by such applications as AddressBook, iPhoto, Mail, and the Finder. By the end of this chapter you will have all the information required to integrate your own application's UI into the Time Machine interface in the same manner accomplished by Apple.

5.1 HERE BE DRAGYNS

The terms and conditions for the Mac App Store prohibits the use of Private APIs. This means that you will **not** be able to sell an application which uses the techniques described in this chapter through the Mac App Store. You could probably put it into a plugin for such an app which you distribute through other means, but I can't guarantee that there won't be repercussions.

In other words: *tread carefully.*

5.2 Private APIs

5.2.1 What's a Private API?

Any large system and SDK provides both *Public APIs* and *Private APIs*. The main and most obvious distinction between the two is that Public APIs are documented and available for you to use, while Private APIs have no documentation and are not publicly mentioned at all.

The more nuanced description is that every programming system is built upon many hundreds of functions and objects, and only a portion are prepared for public consumption and tested thoroughly enough that they can be relied upon to function in a consistent way. These few are designated Public APIs, and they are the gateways to the internal functionality of the system. The underlying functionality is represented by Private APIs. Complex subsystems will be designed and exported only for the benefit of other APIs or applications.

Frequently on Mac OS X these private APIs are encapsulated in *Private Frameworks*. These are standard frameworks (albeit without header files) located in `/System/Library/PrivateFrameworks`. Most if not all public frameworks use these to implement at least part of their functionality. Some of these private frameworks eventually become public— OpenDirectory.framework, URLMount.framework, and DiskArbitration.framework are three which have done just that— in fact, look inside the PrivateFrameworks folder and you'll see symbolic links to their new public location.

In larger public frameworks, such as CoreServices, CoreFoundation, Foundation or AppKit, there is frequently another private-to-public API progression. Many new AppKit features in each release of OS X can actually be found in earlier versions of that framework or in Apple applications. Many items written specifically for certain uses are often cleaned up and moved into a public framework for anyone to use. One example would be the segmented controls used by the Mail application. Others simply exist quietly without header files in one or more OS releases— the CoreText framework (a sub-framework of ApplicationServices) was introduced in OS X 10.4, but the API was only made public in 10.5. The same is true of the Objective-C garbage-collection system.

In our case, the API we're going to use is part of the private **Backup.framework**, which contains the implementation of the Time Machine backup and restore engine, as well as the user interface.

5.2.2 Private API Introspection

Our first task, then, is to figure out just what APIs are there for us to find. For this we turn to the Terminal and the **nm** command-line tool installed as part of the Xcode Tools. The purpose of this tool is to display all the symbol table entries in a binary file, for both imported and exported symbols, for functions and data, whether publicly visible or not.[1]

A varied sampling of the `nm` tool's output can be seen in Listing 5.1, in this instance taken from the CoreFoundation framework.

Listing 5.1: Example `nm` Command Output

```
000000000011da40 t _CFBasicHashCopyDescription
0000000000002f80 T _CFBasicHashCreate
                 U _NSStartSearchPathEnumeration
00000000001794c0 s ___CFBitVectorClass
000000000017dff8 S ↩
      _NSStreamDataWrittenToMemoryStreamKey
00000000001ad100 b __AlternatePlatformLen
00000000001ac478 d ___CFAppleLanguages
00000000001ac470 D ___CFArgStuff
```

The output consists of three columns: the first contains the *value* of the symbol (typically an address), the second indicates its *type*, and the third its *name*. Note that the names all begin with an extra underscore when compared to the corresponding C source code— this is because the C compiler front-end parser adds this underscore when generating assembler code as a matter of convention.

The types shown by `nm` are represented by single letters. These letters are uppercase for exported (externally-linked) symbols, and lowercase for local-only symbols. The different types are:

[1] The Mach-O binary file format allows symbols to be marked as linkable from other binaries or only between objects in the current linker unit. The former are created by using the (implied) `extern` keyword in declarations, while the latter are created using `__private_extern__`.

- **U — Undefined**
  A reference to an imported symbol whose value is computed and inserted by the dynamic loader at runtime.
- **A — Absolute**
  An absolute value which cannot change.
- **T — Text section symbol**
  A symbol present within the `__TEXT,__text` section of the binary, typically a function. Its value is an address.
- **D — Data section symbol**
  A symbol present within the `__DATA` segment of the binary. Commonly a constant value or a global variable. its value is an address.
- **B — BSS section symbol**
  A symbol present within the `__DATA,__bss` section. An uninitialized static variable. Its value is an address.
- **C — A common symbol**
  Used for debugger symbol-table entries only. Its value is an address.
- **S — Generic symbol**
  A symbol residing in a section other than those listed above. Commonly seen when looking at items residing within the `__OBJC` segment or constant, statically-declared data.

The most commonly-encountered types are shown in Listing 5.1 above, but when we look at the Backup framework we will see that no local variables are defined at all. In this case, the binary has been *stripped*, which refers to the process of removing symbol table entries for all non-exported symbols. That kinda sucks for us, but there's nothing we can do to change it.

Running `nm` on the Backup framework reveals that its public API consists of functions beginning with the prefix **BU**. As a result, we can filter the output to show us only those functions, as demonstrated in Listing 5.2.

Listing 5.2: Filtered Backup.framework `nm` Output

```
% nm Backup | grep ".*T _BU.*" | awk '{print "  "$3}'
  _BUAboutToDeleteSnapshots
  _BUActivatedSnapshot
  _BUChangeTimeMachineTarget
  _BUCopyCurrentTargetOriginalOrExistingParentURL
  _BUCopyCurrentTargetOriginalPath
  _BUCopyCurrentTargetSnapshotURL
  _BUCopySnapshotArray
  _BUDeactivatedSnapshot
  _BUFinishResizingWindow
  _BUInvalidateAllSnapshotImages
  _BURegisterActivateSnapshot
  _BURegisterDeactivateSnapshot
  _BURegisterNavigateForwardOrBackward
  _BURegisterRequestRestoreImages
  _BURegisterRequestRevisionID
  _BURegisterRequestSnapshotImage
  _BURegisterShowChangedItemsOnlyToggled
  _BURegisterStartTimeMachineFromDock
  _BURegisterTimeMachineDismissed
  _BURegisterTimeMachineRestore
  _BURegisterUpdateEntryWindow
  _BUStartResizingWindow
  _BUStartTimeMachine
  _BUTimeMachineAction
  _BUTimeMachineSetRestoreAllowed
  _BUUpdateGenericSnapshotImage
  _BUUpdateSnapshotImage
```

There are other routines you might see, but they all begin with at least two underscores, meaning that they were named with at least one in their source code, indicating that they are internal, and an implementation detail of the methods seen above.

There's not a great deal there. We can see some async-notification-style methods such as `_BUActivatedSnapshot` and `_BUFinishResizingWindow`. We can see some command-style functions like `_BUStartTimeMachine`, and we can see a list of what look like callback registration methods beginning with `_BURegister`.

These on their own aren't a great deal of assistance to us however. If we were to pull out `otool` to disassemble the file, we would see that most of these methods pass on their arguments to multiple local methods.

This makes reverse-engineering an inefficient way of determining the functionality of these functions. However, a better way of learning to use this API is simply to watch it being used, and emulate the behaviour we see there.

5.2.3 Watching the Clients

As mentioned in the introduction to this chapter, the most commonly known Apple applications which use the Backup/Time Machine APIs are Mail, iPhoto, and the Address Book. Of these three, the Address Book application looks the smallest, so we will take a look at that one on the assumption that the code we want to emulate will likely be quite straightforward in nature.

We first run `nm` on the Address Book binary and discover a lot of Objective-C symbols. Filtering the list for items mentioning 'Backup' or 'TimeMachine' reveals that it is indeed referencing the Backup API. However the binary has been stripped, leaving us without any clues as to where we might find those references in the code. However, Objective-C is a dynamic language using message-passing, so all function information needs to be encoded in a reloadable format, making it relatively easy to reconstruct class details from a binary file. Our friend in this regard is the venerable 'class-dump' command-line tool.

Class Dump

Originally written by Steve Nygard in 1997, *class-dump* is a command-line tool which will read the contents of the `__OBJC` segment of a Mach-O binary and write out the details in the form of Objective-C header files. It is chiefly used these days for the same purpose to which we will put it in this chapter: reverse-engineering the headers of private Objective-C APIs. It has the capability to output all sorts of additional information inside the headers as well, such as function implementation addresses and member variable offsets. By default it writes to standard output, but the `-H` flag will tell it to generate header files in the current directory, or one specified using the `-o` flag.

An additional option for `class-dump` is the `-C` flag, which allows you to provide a regular expression to filter the list of classes output. The regular expression is applied to the class names themselves. When searching for 'Backup' nothing appears, but change that to 'TimeMachine' and we appear to have hit pay-dirt, receiving the classes shown in Listing 5.3.[2]

Listing 5.3: Classes Containing 'TimeMachine' in the Address Book Application

```
@interface ABTimeMachineController
@interface ABTimeMachineRestoreGroupOperation
@interface ABTimeMachineRestoreOperation
@interface ABTimeMachineSnapshotOperation
@interface ABTimeMachineSource
@interface ABTimeMachineSourcesOperation
```

Right up top we see the item we will likely want to investigate further: `ABTimeMachineController`. We also see a number of items whose names end with `Operation`, suggesting that operation queues are in use here. Perhaps we'll start with those, as knowledge of their operations will quite likely help us understand their use within the controller.

5.2.4 Time Machine Operations

The *Restore* operations we shall skip over, as it is concerned with the internals of the Address Book app, and is therefore of little interest to us today.

[2]Note that I am not including the full headers for any of these classes here as they are quite possibly considered Apple Confidential Information. Anyone can easily obtain their own copies using `class-dump` on their local copy of the Address Book application.

Bibliography

[1] Keith Loepere (Editor), *Mach 3 Server Writer's Guide*. Open Software Foundation and Carnegie Mellon University, 1992. Downloadable from http://www.rtmach.org/manual/server_writer.pdf

[2] Tim Wood, *Mach Exception Handlers 101*. osx-dev Mailing List at Apple, June 2000. Archived at CocoaBuilder: http://www.cocoabuilder.com/archive/cocoa/35756-mach-exception-handlers-101-was-re-ptrace-gdb.html#35756

[3] *Tips and Tricks for Mac Management*. Apple Inc., May 2009. Downloadable from http://images.apple.com/education/docs/Apple-ClientManagementWhitePaper.pdf