7

The Alpha and the Omega — launchd

When you power on your Mac or i-Device, the boot loader (OS X: EFI, iOS: iBoot), described in the previous chapter is responsible for finding the kernel and starting it up. The kernel boot is described in detail in Chapter 7. The kernel, however, is merely a service provider, not an actual application. The user mode applications are those which perform the actual work in a system, by building on kernel primitives to provide the familiar user environment rich with files, multimedia, and user interaction. It all has to start somewhere, and in OS X and iOS — it starts with launchd.

LAUNCHD

launchd is OS X's and iOS's idea of what other UN*X systems call *init*. The name may be different, but the general idea is the same: It is the first process started in user mode, which is responsible for starting — directly or indirectly — every other process in the system. In addition, it has OS X and iOS idiosyncratic features. Even though it proprietary, it still falls under the classification of Darwin, and so it is fully open source[1].

Starting launchd

launchd is started directly by the kernel. The main kernel thread, which is responsible for loading the BSD subsystem, spins off a thread to execute the `bsdinit_task`. The thread assumes PID 1, with the temporary name of "init," a legacy of its BSD origins. It then invokes `load_init_program()`, which calls the `execve()` system call (albeit from kernel space) to execute the daemon. The name — `/sbin/launchd` — is hard coded as the variable `init_program_name`.

The daemon is designed to be started in this way, and this way only; It cannot be started by the user. If you try to do so, it will complain, as shown in Listing 7-1.

LISTING 7-1: Attempting to start launchd will result in failure

```
root@Minion (/)# /sbin/launchd
launchd: This program is not meant to be run directly.
```

Although launchd cannot be started, it can be tightly controlled. The `launchctl(1)` command may be used to interface with launchd, and direct it to start or stop various daemons. The command is interactive, and has its own help.

launchd is usually started with no arguments, but does optionally accept a single command line argument: `-s`. This argument is propagated to it by the kernel, if the latter was started with `-s`, either through its `boot-args`, or by pressing Option-S during startup.

launchd can be started with several logging and debugging features, by creating special dot files in `[/private]/var/db`. The files include `.launchd_log_debug`, `.launchd_log_shutdown` (output to `/var/tmp/launchd-shutdown.log`), and `.launchd_use_gmalloc` (enabling `libGMalloc`, as discussed in Chapter 3). launchd also checks for the presence of the `/AppleInternal` file (on the system root) for some Apple internal logging.

launchd's loading of `libGMalloc` *on iOS (if* `/var/db/.launchd_use` *has been used by the jailbreaker comex in what is now known as the interposition exploit. launchd executes with root privileges, and by crafting a Trojan library, code can be injected into userland root — one step closer to subverting the kernel.*

System-Wide Versus Per-User launchd

If you use `ps(1)` or a similar command on OS X, you will see more than one instance of launchd: The first is PID 1, which was started by the kernel in the manner described previously. If anyone is logged on, there will be another launchd, forked from the first, and owned by the logged in user, shown in Listing 7-2. You may also see other instances, belonging to system users (e.g. spotlight - uid 89).

LISTING 7-2: Two instances of launchd

```
morpheus@ergo (/)$  ps -ef | grep sbin/launchd
    0     1     0   0   6:32.43 ??         6:37.98 /sbin/launchd
  501    95     1   0   0:06.44 ??         0:11.07 /sbin/launchd
```

The per-user launchd is executed whenever a user logs in, even remotely over SSH (though once per logged in user). On iOS there is only one instance of launchd, the system-wide instance.

It is impossible to stop the system-wide launchd (PID 1). In fact, launchd is the only immortal process in the system. It cannot be killed, and that makes sense. There is absolutely no reason to terminate it. In most UN*X, if the init process dies unexpectedly the result is a kernel panic. launchd is also the last process to exit, when the system is shut down.

Daemons and Agents

The core responsibility of launchd is, as its name implies, launching other processes, or jobs, on a scheduled or on-demand basis. launchd makes a distinction between two types of background jobs:

- Daemons are, like the traditional UNIX concept, background services that normally have no interaction with the user. They are started automatically by the system, whether or not any users are logged on.
- Agents are special cases of daemons that are started only when a user logs on. Unlike daemons, they may interface with the user, and may in fact have a GUI.
- iOS does not support the notion of a user login, which is why it only has LaunchDaemons (though an empty `/Library/LaunchAgents` does exist).
- Both daemons and agents are declared in their individual property list (`.plist`) files. As described in Chapter 2, these are commonly XML (in OS X) or binary (in iOS). A detailed discussion of the valid plist entries in the verbose man page — `launchd.plist(5)`, though it should be noted the man page does leave out a few undocumented keys. The rest of this chapter demonstrates the plist format through various examples. The complete list of job keys (including useful keys for sandboxing jobs) can be found in launchd's `launch_priv.h file.`

The list of daemons and agents can be found in the locations noted in Table 7-1.

TABLE 7-1: Launch Daemon locations

DIRECTORY	USED FOR
`/System/Library/LaunchDaemons`	Daemon plist files, primarily those belonging to the system itself.
`/Library/LaunchDaemons`	Daemon plist files, primarily third party.
`/System/Library/LaunchAgents`	Agent plist files, primarily those belonging to the system itself.
`/Library/LaunchAgents`	Other agent plist files, primarily third party. Usually empty.
`~/Library/LaunchAgents`	User-specific launch agents, executed for this user only.

launchd uses the `/private/var/db` directory for its runtime configuration, creating `com.apple.launchd[.peruser.%d]` files for runtime override and disablement of daemons.

The Many Faces of launchd

launchd is the first process to emerge to user mode. When the system is at its nascent stage, it is (briefly) the only process. This means that virtually every aspect of system startup and function is either directly or indirectly dependent on it. In OS X and iOS, launchd serves multiple roles, which in other UN*X are traditionally delegated to several daemons.

init

The first, and chief role played by launchd is that of the daemon init. The job description of the latter involves setting up the system by spawning its myriad daemons, then fading to the background, and ensuring these daemons are alive. If one dies, launchd can simply respawn it.

Unlike traditional init, however, the launchd implementation is somewhat different, and considerably improved, as shown in Table 7-2:

TABLE 7-2: init vs. launchd

RESPONSIBILITY	TRADITIONAL INIT	LAUNCHD
Function as PID 1, great ancestor of all processes	init is the first process to emerge into user mode, and forks other processes (which in turn may fork others). Resource limits it sets for itself are inherited by all of its descendants.	Same. launchd also sets Mach exception ports, which are used by the kernel internally to handle exception conditions and generate signals (see Chapter 8).
Support "run levels"	Traditional init supports run levels: 0 – poweroff 1 – single user 2 – multi-user 3 – multi-user + NFS 5 – halt 6 – reboot	launchd does not recognize run levels and allows only for individual per-daemon or per-agent files. There is, however, a distinction for single-user mode.
Start system services	init runs services in order, per files listed in `/etc/rc?.d` (corresponding to run level), in lexicographic order.	launchd runs both system services (daemons), and per-user services (agents).
System service specification	init runs services as shell scripts, unaware and oblivious to their contents.	launchd processes property list files, with specific keywords.
Restart services on exit	init recognizes the `respawn` keyword in `/etc/inittab` for restart.	launchd allows a KeepAlive key in the daemon or agent's property list.
Default user	Root.	Root, but launchd allows a username key in the property list.

Per-User Initialization

Traditional UN*X has no mechanism to run applications on user login. Users must resort to shell and profile scripts, but those quickly get confusing since each shell uses different files, and not all shells are necessarily login shells. Additionally, in a GUI environment it is not a given that a shell would be started, at all (as is indeed the case with most OS X users, who remain unaware of the `Terminal.app`).

By using LaunchAgents, launchd enables per-user launching of specific applications. Agents can request to be loaded by default in all sessions, or only in GUI sessions, by specifying the `LimitLoadToSessionType` key with values such as `LoginWindow` or `Aqua`, or `Background`.

atd/crond

UN*X traditionally defines two daemons — `atd` and `crond` — to run scheduled jobs, as in executing a specified command at a given time. The first daemon, `atd`, serves as the engine allowing the `at(1)` command for one-time jobs, whereas the second, `crond`, provides recurring job support.

Apple is gradually phasing out `atd` and `crond`. The `atd` is no longer a stand-alone daemon, but is now started by launchd. This service, defined in `com.apple.atrun.plist`, (shown in Listing 7-3) is usually disabled:

LISTING 7-3: The com.apple.atrun.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>com.apple.atrun</string>
    <key>ProgramArguments</key>
    <array>
            <string>/usr/libexec/atrun</string>
    </array>
                                                  launchd starts atrun(8) every 30
    <key>StartInterval</key>                      seconds, if enabled
    <integer>30</integer>

    <key>Disabled</key>                      Disabled by default. Setting Disabled:false
    <true/>                                  (or removing key) enables
</dict>
</plist>
```

The `atrun` plist must be enabled to allow the `at(1)` family of commands to work. Otherwise, it will schedule jobs, but they will never happen (as the author learned the hard way, once relying on it to set a wake-up alarm).

The `crond` service is still supported (in `com.vix.crond.plist`), although launchd has its own set of `StartCalendarInterval` keys to replace it. Apple supplies `periodic(8)` as a replacement. Listing 7-4 shows `com.apple.periodic-daily`, one of the several cron-substitutes (along with `-weekly` and `-monthly`):

LISTING 7-4: com.apple.periodic-daily.plist

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
  "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Label</key>
        <string>com.apple.periodic-daily</string>
        <key>ProgramArguments</key>
        <array>
                <string>/usr/sbin/periodic</string>
                <string>daily</string>
        </array>
        <key>LowPriorityIO</key>
        <true/>
        <key>Nice</key>
        <integer>1</integer>
        <key>StartCalendarInterval</key>
        <dict>
                <key>Hour</key>
                <integer>3</integer>
                <key>Minute</key>
                <integer>15</integer>
        </dict>
        <key>AbandonProcessGroup</key>
        <true/>
</dict>
</plist>
```

In iOS, an alternate method of specifying periodic execution is with the `StartInterval` key. The `/usr/sbin/daily` service, for example, specifies a value of 86,400 seconds (24 hours). Other services, such as `itunesstored` and `softwareupdateservicesd` also use this method.

inetd/xinetd:

In UN*X, inetd (and its successor, xinetd) is used to start network servers. The daemon is responsible for binding the port (UDP or TCP), and — when a connection request arrives — it starts the server on demand, and connects its input/output descriptors (`stdin`, `stderr`, and `stdout`) to the socket.

This approach is highly beneficial to both the network server, and the system. The system does not need to keep the server running if there are no active requests to be serviced, thereby reducing system load. The server, on its part, remains totally agnostic of the socket handling logic, and can be coded to use only the standard descriptors. In this way, an administrator can whimsically reassign port numbers to services, and essentially run any CLI command, even a shell, over a network port.

launchd integrates the inetd functionality into itself*, by allowing daemons and agents to request a particular socket. All the daemon has to do is ask, using a `Sockets` key in its plist. Listing 7-5 shows an example of requesting TCP/IP socket 22, from `ssh.plist`:

* Technically, the inetd functionality is handled by launchproxy(8), also part of the launchd project. The manual page has been promising the two would be merged eventually, but it has yet to happen.

LISTING 7-5: ssh.plist, demonstrating IP socket registration

```
<plist version="1.0">
<dict>
     <key>Disabled</key>
     <true/>

     <key>Label</key>
     <string>com.openssh.sshd</string>

     <key>Program</key>
     <string>/usr/libexec/sshd-keygen-wrapper</string>
     <key>ProgramArguments</key>
     <array>
         <string>/usr/sbin/sshd</string>
         <string>-i</string>
     </array>


     <key>Sockets</key>
     <dict>
        <key>Listeners</key>
        <dict>
           <key>SockServiceName</key>
           <string>ssh</string>

           <key>Bonjour</key>
           <array>
             <string>ssh</string>
             <string>sftp-ssh</string>
           </array>
        </dict>
     </dict>
     <key>inetdCompatibility</key>
     <dict>
          <key>Wait</key>
          <false/>
     </dict>

     <key>StandardErrorPath</key>
     <string>/dev/null</string>

     <key>SHAuthorizationRight</key>
     <string>system.preferences</string>
</dict>
</plist>
```

```
Disabled by default. Setting
Disabled:false (or removing key) enables
```

```
"Label" defines the service
internally (for launchctl(8))
```

```
"Program" specifies path to execute.
Command line arguments are specified in
an array
```

```
SockServiceName refers to /etc/services:
ssh 22/tcp # SSH Remote Login Protocol
```

```
Bonjour advertises the
service(s) over multicast
```

```
inetdCompatibility allows porting from
the legacy inetd.conf (here, "nowait",
allowing multiple instances)
```

```
StandardErrorPath redirects
stderr to /dev/null.
```

Unlike inetd, the socket the daemon is requesting may also be a UNIX domain socket. Listing 7-6, an excerpt from `com.apple.syslogd.plist`, demonstrates this:

LISTING 7-6: com.apple.syslogd.plist, demonstrating UNIX socket registration

```
...
<key>ProgramArguments</key>
        <array>
                <string>/usr/sbin/syslogd</string>
        </array>
        <key>Sockets</key>
        <dict>
                <key>AppleSystemLogger</key>
                <dict>
                        <key>SockPathMode</key>
                        <integer>438</integer>
                        <key>SockPathName</key>
                        <string>/var/run/asl_input</string>
                </dict>
                <key>BSDSystemLogger</key>
                <dict>
                        <key>SockPathMode</key>
                        <integer>438</integer>
                        <key>SockPathName</key>
                        <string>/var/run/syslog</string>
                        <key>SockType</key>
                        <string>dgram</string>
                </dict>
        </dict>
```

The two socket families — UNIX and INET — are not mutually exclusive, and may be specified in the same clause. The previous `syslogd` plist, for example, can easily be modified to allow syslog to accept messages from UDP 514 by adding a `SockServiceName:syslog` key (and optionally appending `-udp_in` and `1` to the `ProgramArguments` array). The iOS daemon `lockdownd` listens in this way on TCP port 62078 and the UNIX socket `/var/run/lockdown.sock`.

mach_init

True to its NEXTStep origins and before the advent of launchd in OS X 10.4, the system startup process was called `mach_init`. This daemon was actually responsible for later spawning the BSD style init, which was a separate process. The two were fused into launchd, and it has assumed `mach_init`'s little documented, but chief role of the bootstrap service manager.

Mach's IPC services rely on the notion of "ports" (vaguely akin to TCP and UDPs), which serve as communication endpoints. This is described (in great detail) in Chapter 10. For the moment, however, it is sufficient to consider a port as an opaque number that can also be referenced by a fully qualified name. Servers and clients alike can allocate ports, but servers either require some type of locator service to allow clients to find them, or otherwise need to be "well-known."

Enter: the bootstrap server. This server is accessible to all processes on the system, which may communicate with it over a given port — the `bootstrap_port`. The clients can then request, over this port, that the server lookup a given service by its name and match them with its port. (UNIX has a similar function in its RPC portmapper, also known as sunrpc. The mapper listens on a well-known port (TCP/UDP 111) and plays matchmaker for other RPC services)[1].

Prior to launchd, mach_init assumed the role of bootstrap_server. launchd has since taken over this role and claims the port (aptly named bootstrap_port) during its startup. Since all processes in the system are its progeny, they automatically inherit access to the port. bootstrap_port is declared as an extern mach_port_t in <servers/bootstrap.h>.

Servers wishing to register their ports with the bootstrap server can use the port to do so, using functions defined in <servers/bootstrap.h>. These functions (bootstrap_create_server and bootstrap_create_service) are still supported, but long deprecated. Instead, the service can be registered with launchd in the server's plist, and a simpler function — bootstrap_check_in() — remains to allow the server to request launchd to hand over the port when it is ready to service requests:

```
kern_return_t bootstrap_check_in(mach_port_t bp,             // bootstrap_port
                                 const name_t service_name, // name of service
                                 mach_port_t *sp);          // out: server port
```

launchd pre-registers the port when processing the server's plist. The server port is usually ephemeral, but can also be well known if the key HostSpecialPort is added. (This is discussed in more detail in Chapter 10, under "Host Special Ports"). launchd can be instructed to wait for the server's request, as is shown in Listing 7-7. com.apple.windowserver.active will be advertised to clients only after WindowServer checks in with launchd using functions from <launch.h>.

LISTING 7-7: com.apple.WindowServer.plist

```
<?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>Label</key>
        <string>com.apple.WindowServer</string>
        <key>ProgramArguments</key>
        <array>

           <string>/System/Library/Frameworks/ApplicationServices.framework/Frameworks/
            CoreGraphics.framework/Resources/WindowServer</string>
           <string>-daemon</string>
        </array>
        <key>MachServices</key>
        <dict>
                <key>com.apple.windowserver</key>
                <true/>
                <key>com.apple.windowserver.active</key>
                <dict>
```

continues

[1]Readers familiar with Android will note the similarity to its Binder mechanism, which (among other IPC related tasks) also allows system services to be published, albeit using a character device, /dev/binder, rather than a port.

LISTING 7-7 *(continued)*

```
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
        </dict>
</dict>
</plist>
```

Any clients wishing to connect to a given service, can then look up the server port using a similar function:

```
kern_return_t bootstrap_look_up(
                mach_port_t bp,             // always bootstrap_port
                const name_t service_name,  // name of service
                mach_port_t *sp);           // out: server port
```

If the server's port is available and the server has checked in, it will be returned to the client, which may then send and receive messages (using `mach_msg()`, also discussed in Chapter 10). The Mach messages for the bootstrap protocol are defined in the launchd source in `.defs` files, which are pre-processed by the Mach Interface Generator (MIG) (also discussed in Chapter 10). You can view a list of the active daemons using the `bslist` subcommand of `launchctl(1)`. The list prints out a flattened view of the hierarchical namespace of bootstrap servers visible in the current context. The `bstree` subcommand displays the full hierarchical namespace (but requires root privileges). In Lion and later, `bstree` also shows XPC namespaces (discussed later in this chapter).

The bootstrap mechanism is now implemented over launchd's `vproc`, a new library introduced in Snow Leopard, which also provides for the next feature, transactions.

Transaction Support

launchd is smarter than the average init. Unlike init, which can just start or stop its daemons, launchd supports transactions, a useful feature exported by launchd's `vproc`, which daemons can access through the public `<vproc.h>`. Daemons using this API can mark pending transactions by encapsulating them between `vproc_transaction_begin`, which generates a transaction handle, and `vproc_transaction_end` on that handle, when the transaction completes. A transaction-enabled daemon can also indicate the `EnableTransactions` key in its plist, which enables launchd to check for any pending transactions when the system shuts down, the user logs out, or after a specified timeout. If there are no outstanding transactions (the process is *clean*), the daemon will be shot down (with a `kill -9`) instead of gracefully terminated (`kill -15`), speeding up the shutdown or logout process, or freeing system resources after sufficient inactivity.

Resource Limits and Throttling

launchd can enforce self-imposed resource limits on its jobs. A job (daemon or agent) can specify `HardResourceLimits` or `SoftResourceLimits` dictionaries, which will cause launchd to call `setrlimit(2)`. The `Nice` key can be used to set the job's `nice` value, as per `nice(1)`. Additionally, a job can be marked with the `LowPriorityIO` key which causes launchd to call `iopolicysys` (system call #322, discussed in Chapter 14) and lower the job's I/O priority. Lastly, launchd is integrated with iOS's Jetsam mechanism (also known as memorystatus, and discussed in Chapter 14), which can enforce virtual memory utilization limitations, a feature that is especially important in iOS, which has no swap space.

Autorun Emulation and File System Watch

One of Windows' most known (and often annoying) features is autorun, which can automatically start a program when removable media (such as a CD, USB storage, or hard disk) is attached. launchd offers the `StartOnMount` key, which can trigger a daemon to start up any time a file system is mounted. This can not only emulate the Windows functionality, but is actually safer, as the autorun feature in Windows has become a vector for malware propagation. launchd's daemon are run from the permanent file system, rather than the removable one.

launchd can also be made to watch a particular path, not necessarily a mount point, for changes, using the `WatchPaths` or the `QueueDirectories` keys. This is very useful, as it can react in real time to file system changes. This functionality is achieved by listening on kernel events (`kqueues`), as discussed in Chapter 3. Daemons may be further extended to support `FSEvents` as well (described in Chapter 4), by specifying a `LaunchEvents` dictionary with a `com.apple.fsevents.matching` dict of matching cases.

I/O Kit Integration

A new feature in Lion is the integration of launchd with I/O Kit. I/O Kit is the runtime environment of device drivers. Launch daemons or agents can request to be invoked on device arrival by specifying a `LaunchEvents` dictionary containing a `com.apple.iokit.matching` dictionary. For the specifics of I/O Kit and its matching dictionaries, turn to Chapter 19. A high-level example, however, can be seen in Listing 7-8, which shows an excerpt from the `com.apple.blued.plist` launch daemon, which is triggered by the to handle Bluetooth SDP transactions.

LISTING 7-8: com.apple.blued.plist, demonstrating I/O Kit triggers

```
<plist version="1.0">
<dict>
        <key>EnableTransactions</key>
        <true/>
        <key>KeepAlive</key>
        <dict>
                <key>SuccessfulExit</key>
                <false/>
        </dict>
        <key>Label</key>
        <string>com.apple.blued</string>
        <key>MachServices</key>
        <dict>
                <key>com.apple.blued</key>
                <true/>
                <key>com.apple.BluetoothDOServer</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
```

continues

LISTING 7-8 *(continued)*

```
        </dict>
        <key>Program</key>
        <string>/usr/sbin/blued</string>
  <key>LaunchEvents</key>
        <dict>
                <key>com.apple.iokit.matching</key>
                <dict>
                        <key>com.apple.bluetooth.hostController</key>
                        <dict>
                                <key>IOProviderClass</key>
                                <string>IOBluetoothHCIController</string>
                                <key>IOMatchLaunchStream</key>
                                <true/>
                        </dict>
                </dict>
        </dict>
</dict>
</plist>
```

Experiment: Setting up a Custom Service

One of the niftiest features of UNIX inetd was its ability to run virtually any UNIX utility on any port. The combination of the inetd's handling of socket logic on the one hand, and the ability to treat a socket as any other file descriptor on the other, provides this powerful functionality.

This is also possible, if a little more complicated with launchd. First, we need to create a launchd plist for our program. Fortunately, this is a simple matter of copy, paste, and modify, as Listing 7-5 can do just fine if you change the `Label`, `Program`, `ProgramArguments`, and `Sockets` keys to whatever you wish.

But here, we encounter a problem: launchd does allow the running of any arbitrary program in response to a network connection, but supports only the redirection of `stdin`, `stdout`, and `stderr` to files. We want the application's `stdin`, `stdout`, and `stderr` to be connected to the socket that launchd will set up for us. This means the program we launch has to be launchd-aware and request the socket handoff.

To solve this, we need to create a generic wrapper, as is shown in Listing 7-9.

LISTING 7-9: A generic launchd wrapper

```
#include <stdio.h>
#include <sys/socket.h>
#include <launch.h> // LaunchD related stuff
#include <stdlib.h> // for exit, and the like
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h> // for getaddrinfo
#include <fcntl.h>
```

```
#define JOBKEY_LISTENERS "Listeners"
#define MAXSIZE 1024
#define CMD_MAX 80


int main (int argc, char **argv)
{
  launch_data_t checkinReq, checkinResp;
  launch_data_t  mySocketsDict;
  launch_data_t  myListeners;

  int fdNum;
  int fd;
  struct sockaddr sa;
  unsigned int    len = sizeof(struct sockaddr);
  int    fdSession ;


  /* First, we must check-in with launchD.  */
  checkinReq = launch_data_new_string(LAUNCH_KEY_CHECKIN);
  checkinResp = launch_msg(checkinReq);

  if (!checkinResp) {
        // Failed to checkin with launchd - this can only be because we are run outside
        // its context. Print a message and exit
        fprintf (stderr,"This command can only be run under launchd\n");
        exit(2);
    }


  mySocketsDict = launch_data_dict_lookup(checkinResp, LAUNCH_JOBKEY_SOCKETS);

  if (!mySocketsDict)
   { fprintf (stderr, "Can't find <Sockets> Key in plist\n"); exit(1); }

  myListeners = launch_data_dict_lookup(mySocketsDict, JOBKEY_LISTENERS);

  if (!myListeners)
   {fprintf (stderr, "Can't find <Listeners> Key inside <Sockets> in plist\n");
   exit(1);



   fdNum = launch_data_array_get_count(myListeners);
   if (fdNum != 1)  {
        fprintf (stderr,"Number of File Descriptors is %d - should be 1\n", fdNum);
        exit(1);
   }

   // Get file descriptor (socket) from launchd
   fd = launch_data_get_fd(launch_data_array_get_index(myListeners,0));

   fdSession = accept(fd, &sa, &len);

   launch_data_free(checkinResp); // be nice..
```

continues

LISTING 7-9 *(continued)*

```
        // Print to stderr (/var/log/system.log) before redirecting..

        fprintf (stderr, "Execing %s\n", argv[1]);

        dup2(fdSession,0);      // redirect stdin
        dup2(fdSession,1);      // redirect stdout
        dup2(fdSession,2);      // redirect stderr
        dup2(fdSession,255);    // Shells also like FD 255.

        // Quick and dirty example - assumes at least two arguments for the wrapper,
        // the first being the path to the program to execute, and the second (and later)
        // being the argument to the launchd program
        execl(argv[1], argv[1], argv[2], NULL);


        // If we're here, the execl failed.
        close(fdSession);

        return (42);
}
```

As the listing shows, the wrapper uses `launchd_` APIs (all clearly prefixed with `launch_` and defined in `<launch.h>`) to communicate with launchd and request the socket. This is done in several stages:

- **Checking in with launchd —** This is done by sending it a special message, using the `launch_msg()` function. Since checking in is a standard procedure, it's a simple matter to craft the message using `launch_data_new_string(LAUNCH_KEY_CHECKIN)` and then pass that message to launchd.
- **Get our plist parameters —** Once launchd has replied to the check-in request, we can use its APIs to get the various settings in the plist. Note that there are two ways to pass parameters to the launched daemons, either as command-line arguments (the `ProgramArguments` array), or via environment variables, which are passed in an `EnvironmentVariables` dictionary, and read by the daemon using the standard `getenv(3)` call.
- **Get the socket descriptor —** Getting any type of file descriptor is a little tricky, since it's not as straightforward to pass between processes as strings and other primitive data types are. Still, any complexity is well hidden by `launch_data_get_fd`.

Once we have the file descriptor (which is the socket that launchd opened for us), we call `accept()` on it, as any network server would. This will yield a connected socket with our client on the other end. All that's left to do is to use the `dup2()` system call to replace our `stdin`, `stdout`, and `stderr` with the accepted socket, and `exec()` the real program. Because `exec()` preserves file descriptors, the new program receives these descriptors in their already connected state, and its `read(2)` and `write(2)` will be redirected over the socket, just as if it would have called `recv(2)` and `send(2)`, respectively.

To test the wrapper, you will need to drop its plist in `/System/Library/LaunchDaemons` (or another `LaunchDaemons` directory) and use `launchctl(1)` to start it, as shown in Output 7-1. The wrapper in this example was labeled `com.technologeeks.wrapper`, and was placed in an eponymous plist. Note in the output, that `launchctl(1)` isn't the chatty type and no comment implies the commands were successful.

OUTPUT 7-1: Using launchctl(1) to start a LaunchDaemon

```
root@Minion (~)# launchctl
launchd% load /System/Library/LaunchDaemons/com.technologeeks.wrapper.plist
launchd% start com.technologeeks.wrapper
launchd% exit
```

Because the wrapper is intentionally generic, you can specify any program you want, assuming this program uses `stdin`, `stdout`, and `stderr` (which all command line utilities do, anyway). This enables nice backdoor functionality, as you can easily set up a root shell on any port you want. Setting the command line arguments to your wrapper to `/bin/zsh -i` will result in output similar to Output 7-2:

OUTPUT 7-2: Demonstrating a launchd-wrapped root shell

```
root@Minion (~)# telnet localhost 1024 # or whereever you set your SockServiceName
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
zsh# id;
uid=0(root) gid=0(wheel) groups=0(wheel),401(com.apple.access_screensharing),
402(com.apple.sharepoint.group.1),1(daemon),
2(kmem),3(sys),4(tty),5(operator),8(procview),9(procmod),12(everyone),
20(staff),29(certusers),
33(_appstore),61(localaccounts)80(admin),98(_lpadmin),100(_lpoperator),
204(_developer)
zsh: command not found: ^M
zsh# whoami;
root
zsh: command not found: ^M
```

Note that a semicolon must be appended to shell commands. This is because you are working directly over the shell's `stdin`, and not a terminal, so the enter key is sent out as a literal Ctrl-M. The semicolon added terminates the command so the shell can parse it, making the Ctrl-M into a separate, invalid command. A minor annoyance in exchange for remote root capabilities.

LISTS OF LAUNCHDAEMONS

There are an inordinate amount of LaunchDaemons in OS X and iOS. Indeed, many sites devote countless HTML pages and SMTP messages to debating the purpose and usefulness of the daemons and agents, especially in iOS, where unnecessary CPU cycles not only impact performance, but also dramatically shorten battery life. The following section aims to elucidate the purpose of these daemons and agents.

iOS and OS X share some common LaunchDaemons. All plists (and their Mach service entries) have the `com.apple` prefix, and usually run their binaries from `/usr/libexec`. They are shown in Table 7-3:

TABLE 7-3: Daemons common to iOS and OS X

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`DumpPanic` `(CoreServices)`	`DumpPanic`	When kernel boots, collects any leftover panic data from a previous panic. Runs with `RunAtLoad=true`.
`appleprofilepolicyd`	`appleprofilepolicyd`	System profiling. Communicates with profiling kernel extensions. Registers `HostSpecialPort 16`.
`aslmanager`	`---`	Apple system Llog. Runs `/usr/bin/aslmanager`, and sets a WatchPath on `/var/log/asl/SweepStore`.
`Backupd` `(MobileBackup framework)`	`Backupd`	`RunAtLoad = true`.
`chud.chum`		Runs `/Developer/usr/libexec/chum`, the CHUD helper daemon allowing access to privileged kernel interfaces from user mode.
`configd`	`SCNetworkReachability` `Configd`	`KeepAlive = true.`
`AppleIDAuthAgent` `(CoreServices)`	`coreservices.appleid .authentication` `coreservices.appleid .passwordcheck`	Handles AppleID-related requests. Whereas iOS has both services, OS X version only has the second service, which runs with a `–checkpassword` switch.
`cvmsServer`	`cvmsServ`	Internal to OpenGL(ES) framework.
`fseventsd`	`FSEvents`	In OS X, `fseventsd` is run from the CarbonCore framework, which is internal to CoreServices.
`locationd`	`locationd.registration` `locationd.simulation (i)` `locationd.spi (i)` `locationd.synchronous (i)` `locationd.agent (SL)` `locationd.services(SL)`	`Location services.`

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`mDNSResponder`	`mDNSResponder`	Multicast DNS listener. Core part of Apple's "Bonjour."
`mDNSResponderHelper`	`mDNSResponderHelper`	Provides privilege separation for `mDNSResponder`.
`notifyd` `(/usr/sbin)`	`system.` `notification_center`	System notification center: handles kernel and other notifications.
`racoon` `(/usr/sbin)`	`Racoon`	Open source VPNd. Thanks to this daemon iOS5 proved jail-breakable (twice).
`ReportCrash` `(/System/Library/` `CoreServices)`	`ReportCrash.*` (OS X has ReportCrash., iOS has JetSam, SafetyNet, Simu-lateCrash, and StackShot.)	The default crash handler, which intercepts all application crashes. Runs automatically on crash by setting job's Mach exception ports (discussed in Chapter 11).
`sandboxd`	`Sandboxd`	Also uses HostSpecialPort 14.
`securityd`	`Securityd` `SecurityServer (SL)`	Handles key access and authori-zation. Written by Perry the Cynic, apparently. `OnDemand`.
`syslogd`	`system.logger`	Passes messages to ASL via the `asl_input` socket (discussed in Chapter 4).

A list of OS X specific LaunchDaemons (and a host of LaunchAgents), is too large and tedious to fit in these pages, but is maintained on the book's companion website.

iOS launchdaemons

Table 7-4 details some of the daemons specific to iOS, in alphabetical order:

TABLE 7-4: Some of the iOS daemons in /System/Library/LaunchDaemons

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
`accessory_device_arbitrator`	`mobile.accessory_device_arbitrator`	Handles accessories plugged into i-Device, such as docks. Set to respond to events from I/O Kit on the `IOUSBInterface`, so it can be started whenever such an accessory is connected. Formerly `accessoryd`.
`Accountsd (Accounts.framework)`	`accountsd.accountmanager` `accountsd.oauthsigner`	Single sign-on. Runs as mobile.
`Amfid`	`MobileFileIntegrity`	Discouraging any attempt to run unsigned, un-entitled code in iOS. Arch-nemesis of all jailbreakers. Uses `HostSpecialPort` 18.
`Apsd (ApplePushService .framework)`	`Apsd`	Apple Push Service Daemon (the APS private framework). Runs as mobile.
`Assetsd (AssetsLibrary.framwork)`	`PersistentURLTranslator .Gatekeeper` `assetsd.*`	Runs as mobile.
`Atc`	`Atc`	Air traffic controller.
`Calaccessd (EventKit.framework/ Support)`	`Calaccessd`	The EventKit's calendar access daemon. Runs as mobile.
`crash_mover`	`crash_mover`	Moves crashes to `/var/Mobile/Library/Logs`.
`fairplayd.XXX`	`Fairplayd` `Unfreed`	User mode helper for Apple's "FairPlay" DRM. This daemon is hardware specific (the plist contains a `LimitedToHardware` key), with *`XXX`* specifying the board type (e.g., N81 for iPod 4,1).
`Itunesstored (iTunesStore.framework/ Support)`	`iTunesStore.daemon.*` `itunesstored.*`	The iTunes Store server. Mostly known for the app store badge notifications. Runs as mobile.
`Lockbot`	`---`	Listens on `/var/run/lockbot`. Assists in jailing the device.

LAUNCHDAEMON (/USR/LIBEXEC)	MACH SERVICES (COM.APPLE.*)	NOTES
Lockdownd	lockdown.host_watcher	See next section of this chapter.
Mobileassetd	Mobileassetd	Runs with `-t 15`.
mobile.installd	mobile.installd	Runs with `-t 30` as mobile.
mobile.installd .mount_helper	mobile.installd .mount_helper	Mounts the developer image when device is selected for development.
mobile_obliterator	mobile.obliteration	Remotely obliterate (that is, wipe) the device.
Pasteboard (UIKit.framework/ Support/)	UIKit.pasteboardd	Cut/paste support. Runs as mobile. Close relative of OS X's as `pboard(8)`, which is a LaunchAgent (q.v., `pbcopy(1)`, `pbpaste(1)`).
SpringBoard (/System/Library/ CoreServices)	CARenderServer SBUserNotification UIKit.statusbarserver bulletinboard.* chatkit .clientcomposeserver.xpc iohideventsystem smsserver springboard.*	The chief UI of i-Devices. Described in its own section in this chapter.
Twitterd (Twitter.Framework)	twitter.authenticate twitterd.server	Twitter support introduced in iOS 5.
Vsassetsd (VoiceServices .framework/Support)	Vsassetd	Responsible for voice assets. Runs as mobile.

Glancing over the table, you may have noticed two special Daemons in iOS: `SpringBoard` and `lockdownd`. `SpringBoard` is the GUI Shell and is described later in this Chapter. `lockdownd` deserves more detail, and is described next.

lockdownd

`lockdownd` is the arch-nemesis of jailbreakers everywhere, being the user mode cop charged with guarding the jail. It is started by launchd and handles activation, backup, crash reporting, device syncing, and other services. It registers the `com.apple.lockdown.host_watcher` Mach service, and listens on TCP port 62078, as well as the `/var/run/lockdown.sock` UNIX domain socket. It is also assisted by a rookie, `/usr/libexec/lockbot`.

`Lockdownd` is, in effect, a mini-launchd. It maintains its own list of services to start in `/System/Library/Lockdown/Services.plist`, as shown in Listing 7-10.

LISTING 7-10: An excerpt from lockdownd's services.plist

```
<plist version="1.0">
<dict>
        <key>com.apple.afc</key>
        <dict>
                <key>AllowUnactivatedService</key>
                <true/>
                <key>Label</key>
                <string>com.apple.afc</string>
                <key>ProgramArguments</key>
                <array>
                        <string>/usr/libexec/afcd</string>
                        <string>--lockdown</string>
                        <string>-d</string>
                        <string>/var/mobile/Media</string>
                        <string>-u</string>
                        <string>mobile</string>
                </array>
        </dict>
        <key>com.apple.afc2</key>
        <dict>
                <key>AllowUnactivatedService</key>
                <true/>
                <key>Label</key>
                <string>com.apple.afc2</string>
                <key>ProgramArguments</key>
                <array>
                        <string>/usr/libexec/afcd</string>
                        <string>--lockdown</string>
                        <string>-d</string>
                        <string>/</string>
                </array>
</dict>
```

The listing shows an important service — `afc` — which is responsible for transferring files between the iTunes host and the i-Device. This is required in many cases, for synchronization as well as moving crash and diagnostic data. The second instance of the same service (`afc2`) is automatically inserted in the jailbreak process, and differs only in its lack of the `-u mobile` command line argument to the `afc`, which makes it retain its root privileges instead of dropping to the non-privileged user mobile. `lockdownd` (just like `launchd`) runs as root and can drop privileges before running another process if the `UserName` key is specified.

GUI SHELLS

When the user logs in on the console (either automatically or by specifying credentials), the system starts a graphical shell environment. OS X uses the Finder, whereas iOS uses `SpringBoard`, but the two are often more similar than they let on. From launchd's perspective, both `Finder` and `SpringBoard` are just one or two more agents in the collection of over 100 daemons and agents they need to start and juggle. But for the user, these programs constitute the first (and often final) frontier for interaction with the operating system.

Finder (OS X)

Finder is OS X's equivalent of Windows' Explorer: It provides the graphical shell for the user. It is started as a launch agent upon successful login, from the `com.apple.Finder.plist` property list (in `/System/Library/LaunchAgents`)

Finder has dependencies on no less than 30 libraries and frameworks, some of them private, which you can easily display by using `otool(1) -l`. Doing so also reveals a peculiarity: Finder is a rare case of an encrypted binary. OS X supports code encryption, as described in Chapter 4 and detailed further in Chapter 13, but there are fairly few encrypted binaries. Output 4-3 demonstrated using `otool -l` to view the encrypted portion of Finder. Using `strings(1)` or trying to disassemble Finder is, therefore, a vain effort (unless the encryption is defeated, for example by a tool like corerupt, presented in Chapter 12). You can also use GDB to attach to Finder once it is running (yet again, defeating the whole purpose of the binary protection), and trace its threads (usually only three of them).

Finder is so tightly integrated with the system that the very design of the native file system, HFS+, has been built around it. The file and folder data, and indeed the volume data itself, contains special finder information fields. These fields enable many features, such as reopening folder windows in the exact dimensions and location the user placed them last. Finder additionally makes use of extended attributes to store information, such as color labels and aliases. These features are all discussed in Chapter 16 (which is entirely devoted to HFS+).

With a Little Help from My Friends

All the work of supporting the rich GUI can prove overwhelming for any one process, which is why the GUI handling is actually split between several processes, which are all in `/System/Library/CoreServices`.

The `Dock.app` is responsible for the familiar tray of icons usually found at the bottom of the desktop, as its name implies, but also sets the wallpaper (what X would call the "root window"), as can be witnessed when the process is killed. It is assisted by `com.apple.dock.extra`, which connects the UI actions to the Dock action outlets.

The `SystemUIServer.app` is responsible for the menu extras (right hand) side of the status bar, which it loads from `/System/Library/CoreServices/Menu Extras`. Note that there, menu extras may also be created programmatically (using `[NSStatusBar systemStatusBar]` and its `setImage/setMenu` methods), in which case these extras are the responsibility of the app which created them.

Due to their important role (and Apple's desire to keep their UI theirs for as long as possible before others "adopt" it), Finder's assistants (as well as other `CoreServices` apps) are also protected binaries.

Experiment: Figuring Out Who Owns What in the GUI

Using a shell (preferably over SSH) and the UNIX `kill(1)` command, you can quickly determine which process owns what part of the GUI. Your options are to either kill the process violently (using `kill -9`) or just pause the process (using `kill -STOP` and `kill -CONT`). Doing so on the various processes — Finder, Dock and SystemUIServer — will either briefly make their UI assets disappear (if killed, until the processes are automatically restarted by launchd) or hang with the spinning beachball of death (as long as the processes are stopped) or a "fast forward" effect (when the processes are resumed, and all the queued UI messages are delivered). Menu extras created by apps will be unaffected by SystemUIServer's suspension or premature demise.

You might want to use `killall(1)` instead of kill, as it will send a signal by name, rather than by PID. If you use it this way to kill the same process repeatedly, launchd throttles the processes, which after a few seconds are respawned.

SpringBoard (iOS)

What Finder is to OS X, SpringBoard is for iOS. In iOS the system need not logon, so SpringBoard is started automatically, to provide the familiar icon based UI of the system. This UI has served as the inspiration to Lion's LaunchPad, which uses the same GUI concepts and is essentially a back port of SpringBoard into OS X — a fact that is evident as some SpringBoard-named files can be found in LaunchPad binary (which is technically part of the dock). Much like its OS X GUI counterpart (Finder), SpringBoard is loaded from `/System/Library/CoreServices/`.

All by Myself (Sort of)

Unlike Finder, SpringBoard handles almost everything by itself, and there are only a few loadable bundles in the CoreServices directory. Finder's 30 dependencies are dwarfed by SpringBoard, which has about 80, as you can see with `otool -l`, which will also reveal that SpringBoard is (surprisingly) an unprotected binary.

SpringBoard nonetheless does turn to additional bundles for certain tasks. `/System/Library/SpringBoardPlugins` contains three types of loadable bundles (as of iOS 5):

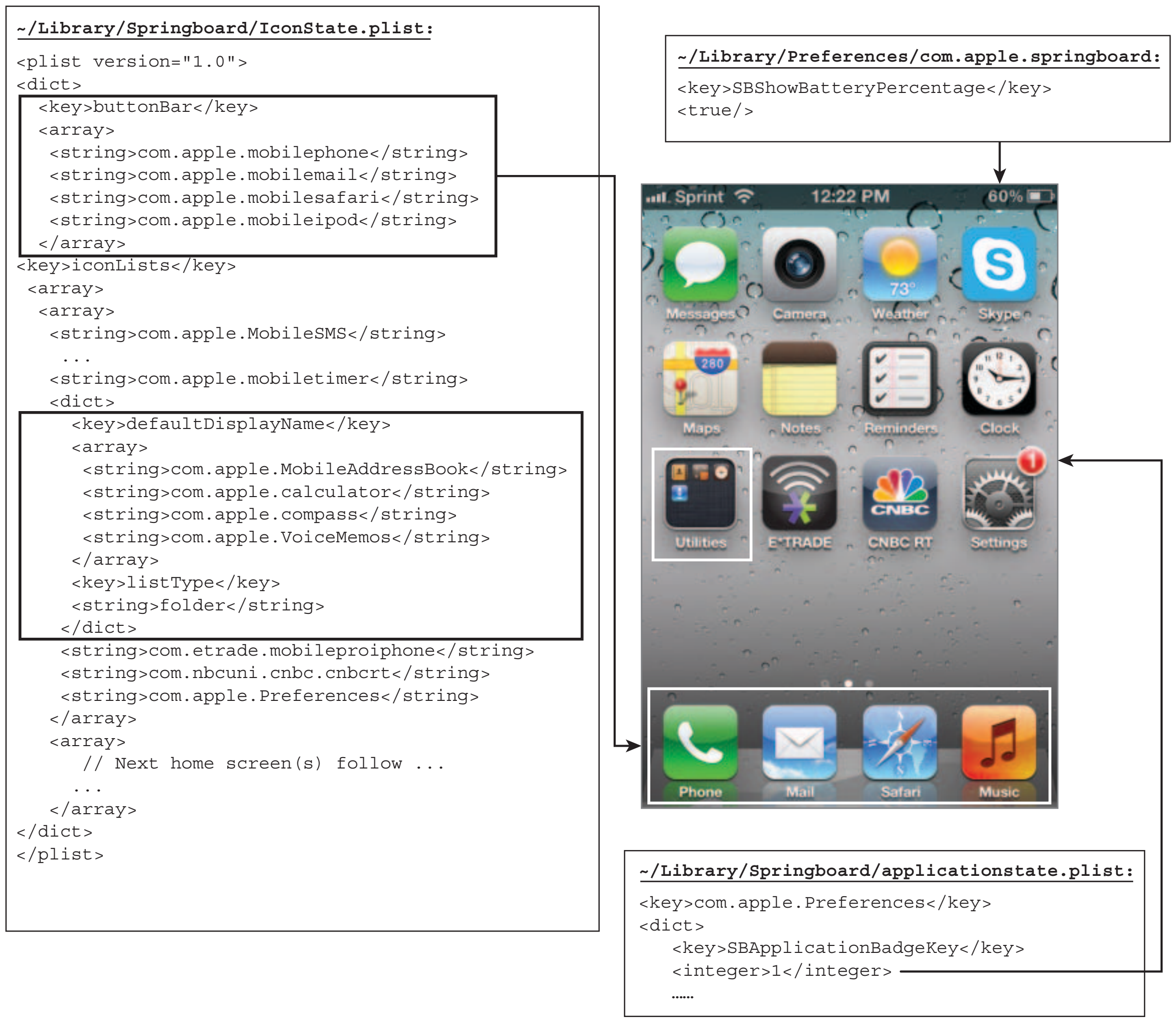
- **`lockbundle`** — Lock bundles provide lock screen functionality. The `NowPlayingArtLockScreen.lockbundle` is responsible for providing the lock screen when the music player (Music~iphone or MobileMusicPlayer) is active and the screen is locked. The `PictureFramePlugin` shows pictures from the user's photo library. The iPhone also has a bundle for `VoiceMemosLockScreen` (to show voice messages and missed call indicators)
- **`servicebundle`** — Helps SpringBoard with various tasks, such as `ChatKit.servicebundle`, `IncomingCall.servicebundle`, and `WiFiPicker.servicebundle`.
- **`bundle`** — The original extension before iOS 5. Still exists for `NikeLockScreen.bundle` and `ZoomTouch.bundle`.

Creating the GUI

SpringBoard creates its GUI by enumerating the apps in `/Applications /var/mobile/Applications` and displaying icons for them on the i-Device. Icon enumeration is performed automatically when SpringBoard starts. Each app's `Info.plist` is read, and the app is displayed on one of the home screens with the icon specified in its `CFBundleIcons` property, unless it contains the `SBAppTags` key with a `hidden` array entry). Examples of hidden apps are Apple's own `DemoApp.app`, `iOS Diagnostics.app`, `Field Test.app`, `Setup.app`, and `TrustMe.app`.

> *iOS devices start* `Setup.app` *when first launched to configure the device, register, and activate it. This has been rumored to annoy certain types of people. A nice way to get past it is to jailbreak the device and boot it (tethered or untethered doesn't matter), then* `ssh` *into it and simply rename (*`mv`*)* `/Applications/Setup.app` *(the new name doesn't matter). Then, restart SpringBoard (*`killall SpringBoard`*), and that setup screen is gone. iTunes will still complain about device registration when syncing, but there are ways to bypass that, as well.*

Icon grouping and the button bar settings are saved to `/var/mobile/Library/SpringBoard/IconState.plist`, with general home screen settings (as well as ringtones and other audio effects) in `/var/mobile/Library/Preferences/com.apple.springboard`. A third file, `applicationstate.plist`, controls application settings like badges. Figure 7-1 shows the mapping between the files and the home screen.



~/Library/Springboard/IconState.plist:

```
<plist version="1.0">
<dict>
  <key>buttonBar</key>
  <array>
    <string>com.apple.mobilephone</string>
    <string>com.apple.mobilemail</string>
    <string>com.apple.mobilesafari</string>
    <string>com.apple.mobileipod</string>
  </array>
<key>iconLists</key>
 <array>
  <array>
    <string>com.apple.MobileSMS</string>
     ...
    <string>com.apple.mobiletimer</string>
    <dict>
      <key>defaultDisplayName</key>
      <array>
       <string>com.apple.MobileAddressBook</string>
       <string>com.apple.calculator</string>
       <string>com.apple.compass</string>
       <string>com.apple.VoiceMemos</string>
      </array>
      <key>listType</key>
      <string>folder</string>
     </dict>
     <string>com.etrade.mobileproiphone</string>
     <string>com.nbcuni.cnbc.cnbcrt</string>
     <string>com.apple.Preferences</string>
    </array>
    <array>
       // Next home screen(s) follow ...
       ...
    </array>
</dict>
</plist>
```

~/Library/Preferences/com.apple.springboard:

```
<key>SBShowBatteryPercentage</key>
<true/>
```

~/Library/Springboard/applicationstate.plist:

```
<key>com.apple.Preferences</key>
<dict>
   <key>SBApplicationBadgeKey</key>
   <integer>1</integer>
   ……
```

FIGURE 7-1: SpringBoard's files and how they lay out the iOS home screen.

Experiment: Unhiding (or Hiding) an iOS App

It's a simple matter to hide or unhide apps on a jailbroken device. All it takes is editing the App's `Info.plist` and toggling the `SBAppTags` key. This is demonstrated in this simple experiment. You can use the method here to unhide or hide any app you wish.

For the app you choose, take the `Info.plist` and copy it to `/tmp`. Then, convert it to the more readable XML format (or, if you prefer, JSON) using `plutil(1)`. Edit the file to either add or remove the `SBAppTags` key with an array, containing a single string value of '`hidden`'. Finally, restart SpringBoard.

Performing the sequence of operations described here on DemoApp, we would have the sequence shown in Output 7-3:

OUTPUT 7-3: Toggling the visibility of an iOS app

```
root@padishah (/)# cp /Applications/DemoApp.app/Info.plist /tmp
root@padishah (/)# plutil -convert xml1 /tmp/Info.plist
Converted 1 files to XML format
root@padishah (/)# cat /tmp/Info.plist
…
     <key>SBAppTags</key>
    <array>                                   Add or remove this value
        <string>hidden</string>
     </array>
…

root@padishah (/)# plutil -convert binary1 /tmp/Info.plist
Converted 1 files to binary format


root@padishah (/)# cp /tmp/Info.plist /Applications/DemoApp.app/
root@padishah (/)# killall SpringBoard
```

Handling the UI

Finder and SpringBoard are both in charge of presenting the UI, but Springboard's responsibilities extend above and beyond. SpringBoard is apparently responsible for every type of action in iOS. Even if it is not the foreground application, if it is stopped (by signal) no UI events get to the active app, and when it is continued all the events queued are delivered to the app.

Springboard is a multithreaded application. It has far more threads than Finder. Apple's developers were kind enough to name some of them (using the `pthread_setname_np`). The names reveal two Web related threads (WebCore and WebThreads), at least two belonging to `coremedia.player`, one for the WiFiManager callbacks (responsible for the WiFi indicator on the status bar), and three or more threads used for CoreAnimation. Debugging the process requires getting past a system watchdog, which reboots the system if SpringBoard is not responsive for more than a few minutes.

More information can be gleaned from Springboard's launchd registration, i.e., the `com.apple.SpringBoard.plist` entry in `/System/Library/LaunchDaemons`, shown in Listing 7-11. Since all

Mach port registrations go through launchd, this lists the (many) ports which SpringBoard requests launchd to register.

LISTING 7-11: SpringBoard's registered Mach ports

```
<plist version="1.0">
<dict>
        <key>EmbeddedPrivilegeDispensation</key>
        <true/>
        <key>HighPriorityIO</key>
        <true/>
        <key>KeepAlive</key>
        <true/>
        <key>Label</key>
        <string>com.apple.SpringBoard</string>
        <key>MachServices</key>
        <dict>
                <key>PurpleSystemEventPort</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.CARenderServer</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.SBUserNotification</key>
                <true/>
                <key>com.apple.UIKit.statusbarserver</key>
                <true/>
                <key>com.apple.bulletinboard.observerconnection</key>
                <true/>
                <key>com.apple.bulletinboard.publisherconnection</key>
                <true/>
                <key>com.apple.bulletinboard.settingsconnection</key>
                <true/>
                <key>com.apple.chatkit.clientcomposeserver.xpc</key>
                <true/>
                <key>com.apple.iohideventsystem</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.smsserver</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
```

continues

LISTING 7-11 *(continued)*

```
                </dict>
                <key>com.apple.springboard.UIKit.migserver</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.alerts</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.appstatechanged</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.backgroundappservices</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.blockableservices</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.processassertionservices</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.processinvalidation</key>
                <dict>
                        <key>HideUntilCheckIn</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.remotenotifications</key>
                <dict>
                        <key>ResetAtClose</key>
                        <true/>
                </dict>
                <key>com.apple.springboard.services</key>
        <dict>
                       <key>HideUntilCheckIn</key>
                       true/>
                       <key>ResetAtClose</key>
                       <true/>
                <key>com.apple.springboard.watchdogserver</key>
                      <true/>
             </dict>
             <key>ProgramArguments</key>
             <array>
                   <string>/System/Library/CoreServices/SpringBoard.app/SpringBoard</string>
             </array>
             <key>ThrottleInterval</key>
             <integer>5</integer>
             <key>UserName</key>
             <string>mobile</string>
      </dict>
      </plist>
```

Chief among all these ports is the `PurpleSystemEventPort`, which handles the UI events as `GSEvent` messages. This is understandably undocumented by Apple, but has been reverseengineered[2]. The main thread in Springboard calls processes `GSEventRun()`, which is the CF RunLoop that handles the UI messages. The other threads are in similar run loops over the other Mach ports in Springboard, but due to the opaque nature of these ports, it's difficult to tell which thread is on which port without the right symbols.

XPC (LION AND IOS)

XPC is a set of lightweight interprocess communication primitives first introduced in Lion and iOS 5. XPC is fairly well documented in Apple Developer[3]. It is also tightly integrated with the Grand Central Dispatcher (GCD). XPC enables a developer to break down applications into separate components. This improves both application stability and security, as vulnerable (or unstable) functionality can be contained in an XPC service, which is managed externally — another responsibility happily assumed by launchd.

Just as with its own `LaunchDaemons`, launchd takes on the tasks of starting XPC services on demand, watching over them (restarting on crash), and terminating them (the hard way, with a `kill -9`) when they are done or idle. The launchd uses `xpcd(8)`, `xpchelper(8)`, and `xpcproxy(8)` to assist with the XPC services. It maintains XPC services alongside standard Mach services, in separate XPC domains — per-user, private, and singleton. This can be seen in the output of `launchctl`'s `bstree` subcommand, as shown in Output 7-4:

OUTPUT 7-4: XPC Service Domains

```
root@Simulacrum (/)# launchctl bstree | grep Domain
com.apple.xpc.domain.com.apple.dock.[231] (XPC Private Domain)/
    com.apple.xpc.domain.Dock[175] (XPC Private Domain)/
    com.apple.xpc.domain.peruser.501 (XPC Singleton Domain)/
    com.apple.xpc.domain.imagent[214] (XPC Private Domain)/
    com.apple.xpc.domain.com.apple.audio[203] (XPC Private Domain)/
    com.apple.xpc.domain.peruser.202 (XPC Singleton Domain)/
    com.apple.xpc.domain.coreaudiod[108] (XPC Private Domain)/
    com.apple.xpc.system (XPC Singleton Domain)/
        ...
```

XPC services and client applications link (either directly or through Cocoa) with `libxpc.dylib`, which provides the various C-level XPC primitives (such as Mountain Lion's `NSXPCConnection`). The library remains closed source at the time of this writing, but Apple does provide the `<xpc/*>` includes which expose the APIs, whose internals are discussed in this section. XPC also relies on the private frameworks of `XPCService` and `XPCObjects`. The former handles runtime aspects of services, and the latter provides encoding and decoding services for XPC objects. iOS contains a third private framework, `XPCKit`.

XPC Object Types

XPC wraps and serializes various datatypes in a manner akin to the `CoreFoundation` framework. `<xpc/xpc.h>` defines the object and data types supported by XPC, shown in Table 7-5. The type names are `#defined` as `XPC_TYPE_`*`typename`* macros wrappings pointers to the corresponding types in the table, and can be instantiated with `xpc_`*`typename`*`_create` functions. Objects can be retrieved from messages in most cases using `xpc_`*`typename`*`_get_value`. Two special object types are dictionaries and arrays, which serve as containers for other object types (which may be created in or accessed from from them using `xpc_[array|dictionary]_[get|set]_`*`typename`*.

TABLE 7-5: XPC Object and data types

TYPE	REPRESENTS
`connection`	An XPC connection, over which messages can be sent and received. A connection can be created using `xpc_connection_create()`, specifying an anonymous or named connection, or from a given endpoint, through a call to `xpc_connection_create_from_endpoint()`.
`endpoint`	Serializable form of a connection. Effectively a connection factory.
`null`	A null object reference (constant) for comparisons.
`bool`	A Boolean.
`true/false`	Boolean true/false values (constants) for comparisons.
`int64/uint64`	Signed/Unsigned 64-bit integers.
`double`	Double precision floats.
`date`	Date intervals (UNIX time). Can be instantiated from the present time by a call to `xpc_date_create_from_current`.
`data`	Array of bytes. The recipient can obtain a pointer to the data by calling `xpc_data_get_bytes_ptr`.
`string`	Null terminated C-String (wraps char *). Strings may be created with a format string, and even with variable arguments (similar to `vsprintf(3)`). The recipient can obtain a pointer to the string by calling `xpc_string_get_string_ptr`.

TYPE	REPRESENTS
`uuid`	Universally Unique Identifier. The recipient can obtain the UUID by a call to `xpc_uuid_get_bytes`.
`fd`	File descriptor. The descriptor can be used by the client by calling `xpc_fd_dup`.
`shmem`	Shared memory. The shared memory can be mapped into the receipient's address space by calling `xpc_shmem_map`.
`array`	Indexed array of XPC objects. An array may contain any number of other object types, which may be added to it or retrieved from it using `xpc_array_[get|set]_`*`typename`*.
`dictionary`	Associative array of XPC objects. A dictionary may contain any number of other object types, which may be added to it or retrieved from it using `xpc_dictionary_[get|set]_`*`typename`*.
`error`	Error objects. Used for returning errors. Cannot be instantiated by clients.

Any of the XPC objects can be handled as an opaque `xpc_object_t`, and manipulated by functions described in `xpc_object(3)`. These include `xpc_retain/release`, `xpc_get_type` (which returns one of the `XPC_TYPE`s corresponding to Table 7-5), `xpc_hash` (used to provide a hash value of an object for array indexing), `xpc_equal` (for comparing objects) and `xpc_copy`.

XPC Messages

Objects may be sent or received in messages. Messages are sent using one of several functions from `<xpc/connection.h>`, as shown in Table 7-6:

TABLE 7-6: XPC Messaging functions in <xpc/connection.h>

FUNCTION	USAGE
`xpc_connection_send_message` `(xpc_connection_t` *`connection`*`, xpc_object_t` *`message`*`);`	Send *`message`* asynchronously on *`connection`*.
`xpc_connection_send_barrier` `(xpc_connection_t` *`connection`*`, dispatch_block_t` *`barrier`*`);`	Execute *`barrier`* block after last message is sent on *`connection`*.
`xpc_connection_send_message_with_reply` `(xpc_connection_t` *`connection`*`, xpc_object_t` *`message`*`, dispatch_queue_t` *`replyq`*`, xpc_handler_t` *`handler`*`);`	Send message, but also asynchronously execute *`handler`* in dispatch queue *`replyq`* when a reply is received.

continues

TABLE 7-6 *(continued)*

`xpc_object_t` **`xpc_connection_send_message_with_reply_sync`** `(xpc_connection_t` *`connection`*`, xpc_object_t` *`message`*`);`	Send *`message`*, blocking until a reply is received, and return reply as the `xpc_object_t` return value

By default, messages are sent asynchronously, and are handled by dispatch queues (i.e., GCD), as shown in Figure 7-2. By using *barriers*, the programmer may provide a block to be executed when all the messages on a particular connection have been sent. Messages may expect replies, which are again asynchronous, though the `_reply_sync` function may be used to block until a message is received.

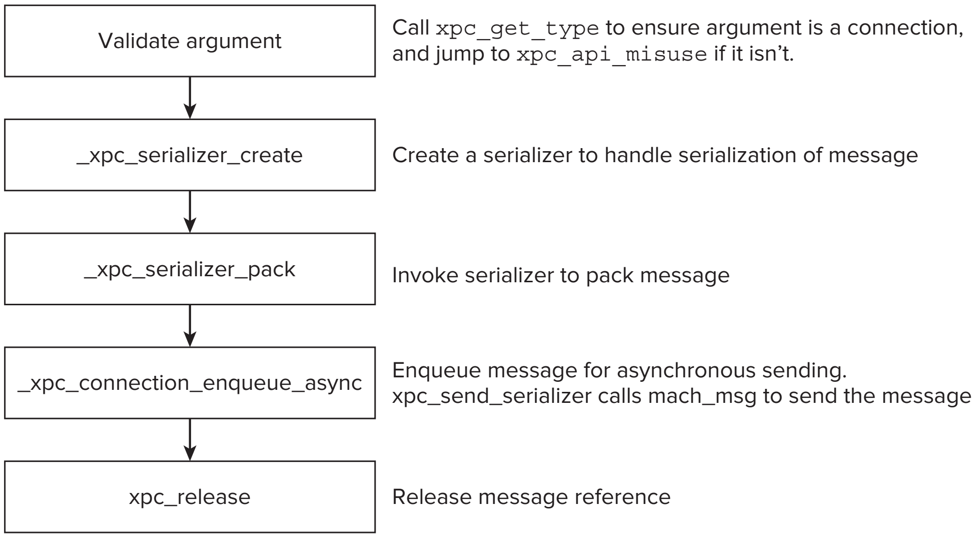

Validate argument
Call xpc_get_type to ensure argument is a connection, and jump to xpc_api_misuse if it isn't.
_xpc_serializer_create
Create a serializer to handle serialization of message
_xpc_serializer_pack
Invoke serializer to pack message
_xpc_connection_enqueue_async
Enqueue message for asynchronous sending. xpc_send_serializer calls mach_msg to send the message
xpc_release
Release message reference

FIGURE 7-2: Flow of xpc_connection_send_message

XPC messages are implemented over Mach messages and make use of the Mach Interface Generator (MIG) facility, which provides the `xpc_domain` subsystem. This subsystem contains messages to check in, load, or add services, and get the name of a service, similar to the bootstrap protocol described earlier in this chapter (XPC can be considered a subset of bootstrap, and makes use of it internally). Mach messages and in particular MIG are detailed in Chapter 10.

XPC services

XPC services can be created in Objective-C, or in C/C++. In either case, the services are started by a call to `libxpc.dylib`'s `xpc_main`. C/C++ services' main is just a simple wrapper, which invokes `xpc_main` (declared in `<xpc/xpc.h>`) with the event handler function (`xpc_connection_handler_t`). Objective-C services also call on `xpc_main()`, albeit indirectly through `NSXPCConnection`'s resume method.

The event handler function takes a single argument, an `xpc_connection_t`. (Objective-C wraps this object with `Foundation.framework`'s `NSXPCConnection`.) The XPC connection is treated as an opaque object, with miscellaneous `xpc_connection_*` functions. In `<xpc/connection.h>` used as getters for its properties, and setters for its event handler and target queue. A connection's name, effective UID and GID, PID and Audit Session ID can all be queried.

The normal architecture of an XPC service involves calling `dispatch_queue_create` to create a queue for the incoming messages from the client and using `xpc_connection_set_target_queue` to assign the queue to the connection. The service also sets an event handler on the connection, calling `xpc_connection_set_event_handler` with a handler block (which may wrap a function). The handler is called whenever the service receives a message. A service may create a reply (by calling `xpc_dictionary_create_reply`) and send it.

A well-documented example of XPC is SandBoxedFetch, which is available from Apple Developer[4], alleviating the need for an example in this book.

XPC Property Lists

XPC services are defined in their own bundles, contained in an `XPCServices` subfolder of its parent application or framework. As with all bundles, they have an `Info.plist`, which they use to declare various service properties and requirements:

- The `CFBundlePackageType` property is defined as "XPC!"
- The `CFBundleIdentifier` property defines the name of the `XPCService`. This is set to be the same as the bundle's name.
- The `XPCService` property defines a dictionary, which can specify the `ServiceType` property (`Application`. `User` or `System`), and `RunLoopType` (`dispatch_main` or `NSRunLoop`), which dictates which run loop style `xpc_main()` adopts. The dictionary may also contain the `JoinExistingSession` Boolean property, to redirect auditing to the application's existing audit session.
- The `XPCService` dictionary may be used to specify additional properties, prefixed by an underscore. These include `_SandboxProfile` (which allows the optional specification of a sandbox profile to enforce on the XPC service, as discussed in Chapter 4) and `_AllowedClients`, which can specify the identifiers of applications which are allowed to connect to the service.

SUMMARY

This chapter discussed launchd, the OS X and iOS replacement to the traditional UNIX init. launchd fills many functions in both operating systems: both those of UNIX daemons, and those of Mach. The Mach roles will be discussed further when the concept of Mach messages is elaborated on in Chapter 10.

The chapter ended with a review of the GUI of both OS X (Finder) and iOS (SpringBoard), in as much detail as possible on these intentionally undocumented binaries.

REFERENCES AND FURTHER READING

1. launchd Sources, `http://opensource.apple.com/tarballs/launchd/launchd-392.38.tar.gz` or later.
2. GSEvent iPhone Development Wiki, `http://iphonedevwiki.net/index.php/GSEvent`
3. Apple Developer, "Daemons and Services Programming Guide" `http://developer.apple.com/library/mac/#documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingXPCServices.html`
4. Apple Developer, "Sandboxed Fetch" `http://developer.apple.com/library/mac/#samplecode/SandboxedFetch/`