4

WIRELESS NETWORKS MADE EASY

It is rather tempting to say that on BSD, and OpenBSD in particular, there's no need to "make wireless networking easy," because it already is. Getting a wireless network running is not very different from getting a wired one up and running, but there are some issues that turn up simply because we are dealing with radio waves and not wires. We will look briefly at some of the issues before moving on to the practical steps involved in creating a usable setup.

Once we have covered the basics of getting a wireless network up and running, we'll turn to some of the options for making your wireless network more interesting and harder to break.

A Little IEEE 802.11 Background

Setting up any network interface, in principle, is a two-step process: you establish a link, and then you move on to configuring the interface for TCP/IP traffic.

In the case of wired Ethernet-type interfaces, establishing the link usually consists of plugging in a cable and seeing the link indicator light up. However, some interfaces require extra steps. Networking over dial-up connections, for example, requires telephony steps such as dialing a number to get a carrier signal.

In the case of IEEE 802.11-style wireless networks, getting the carrier signal involves quite a few steps at the lowest level. First, you need to select the proper channel in the assigned frequency spectrum. Once you find a signal, you need to set a few link-level network identification parameters. Finally, if the station you want to link to uses some form of link-level encryption, you need to set the correct kind and probably negotiate some additional parameters.

Fortunately, on BSD systems, all configuration of wireless network devices happens via `ifconfig` commands and options, as with any other network interface.[1] Still, since we are introducing wireless networks here, we need to look at security at various levels in the networking stack from this new perspective.

There are basically three kinds of popular and simple IEEE 802.11 security mechanisms, and we will discuss them briefly over the next sections.

NOTE *For a more complete overview of issues surrounding security in wireless networks, see Professor Kjell Jørgen Hole's articles and slides at* http://www.kjhole.com/ *and* http://www.kjhole.com/Standards/WiFi/WiFiDownloads.html. *For fresh developments in the Wi-Fi field, the Wi-Fi Net News site* (http://wifinetnews.com/archives/cat_security.html) *and The Unofficial 802.11 Security Web Page* (http://www.drizzle.com/~aboba/IEEE/) *are highly recommended.*

MAC Address Filtering

The short version of the story about PF and MAC address filtering is that we don't do it.

A number of consumer-grade, off-the-shelf wireless access points offer MAC address filtering, but contrary to common belief, they don't really add much security. The marketing succeeds largely because most consumers are unaware that it's possible to change the MAC address of essentially any wireless network adapter on the market today.[2]

1. On some systems, the older, device-specific programs such as `wicontrol` and `ancontrol` are still around, but for the most part, they are deprecated and in the process of being replaced with `ifconfig` functionality. On OpenBSD, the consolidation into `ifconfig` has been completed.
2. A quick man page lookup on OpenBSD will tell you that the command to change the MAC address for the interface `rum0` is simply `ifconfig rum0 lladdr 00:ba:ad:f0:0d:11`.

NOTE *If you really want to try MAC address filtering, you could look into using the* `bridge(4)` *facility and the MAC filtering features offered by* `brconfig(8)` *(on OpenBSD 4.6 and earlier) or the bridge-related rule options in* `ifconfig(8)` *(OpenBSD 4.7 and later). We'll look at bridges and some of the more useful ways to use them with packet filtering in Chapter 5.*

WEP

One consequence of using radio waves instead of wires to move data is that it is comparatively easy for outsiders to capture data in transit over radio waves. The designers of the 802.11 family of wireless network standards seem to have been aware of this fact, and they came up with a solution that they went on to market under the name *Wired Equivalent Privacy*, or *WEP*.

Unfortunately, the WEP designers came up with their wired equivalent encryption without actually reading up on recent research or consulting active researchers in the field. So, the link-level encryption scheme they recommended is considered a pretty primitive homebrew among cryptography professionals. It was no great surprise when WEP encryption was reverse-engineered and cracked within a few months after the first products were released.

Even though you can download free tools to descramble WEP-encoded traffic in a matter of minutes, for a variety of reasons, WEP is still widely supported and used. Essentially, all IEEE 802.11 equipment available today has support for at least WEP, and a surprising number offer MAC address filtering, too.

You should consider network traffic protected only by WEP to be just marginally more secure than data broadcast in the clear. Then again, the token effort needed to crack into a WEP network may be sufficient to deter lazy and unsophisticated attackers.

WPA

It dawned on the 802.11 designers fairly quickly that their WEP system was not quite what it was cracked up to be, so they came up with a revised and slightly more comprehensive solution called *Wi-Fi Protected Access*, or *WPA*.

WPA looks better than WEP, at least on paper, but the specification is complicated enough that its widespread implementation was delayed. In addition, WPA has attracted its share of criticism over design issues and bugs. Combined with the familiar issues of access to documentation and hardware, free software support varies. Most free systems have WPA support, but you may find that it is not available for all devices. If your project specification includes WPA, look carefully at your operating system and driver documentation.

And, of course, it goes almost without saying that you will need further security measures, such as SSH or SSL encryption, to maintain any significant level of confidentiality for your data stream.

The Right Hardware for the Task

Picking the right hardware is not necessarily a daunting task. On a BSD system, the following simple command is all you need to enter to see a listing of all manual pages with the word *wireless* in their subject lines.[3]

```
$ apropos wireless
```

Even on a freshly installed system, this command will give you a complete list of all wireless network drivers available in the operating system.

The next step is to read the driver manual pages and compare the lists of compatible devices with what is available as parts or built into the systems you are considering. Take some time to think through your specific requirements. For test purposes, low-end `rum` or `ural` USB dongles will work. Later, when you are about to build a more permanent infrastructure, you may want to look into higher-end gear. You may also want to read Appendix B of this book.

Setting Up a Simple Wireless Network

For our first wireless network, it makes sense to use the basic gateway configuration from the previous chapter as our starting point. In your network design, it is likely that the wireless network is not directly attached to the Internet at large, but the wireless network will require a gateway of some sort. For that reason, it makes sense to reuse the working gateway setup for this wireless access point, with some minor modifications introduced over the next few paragraphs. After all, doing so is more convenient than starting a new configuration from scratch.

NOTE *We are in infrastructure-building mode here, and will be setting up the access point first. If you prefer to look at the client setup first, see "The Client Side" on page 51.*

The first step is to make sure you have a supported card and check that the driver loads and initializes the card properly. The boot-time system messages scroll by on the console, but they are also recorded in the file */var/run/dmesg.boot*. You can view the file itself or use the `dmesg` command to see these messages. With a successfully configured PCI card, you should see something like this:

```
ral0 at pci1 dev 10 function 0 "Ralink RT2561S" rev 0x00: apic 2 int 11 (irq
11), address 00:25:9c:72:cf:60
ral0: MAC/BBP RT2561C, RF RT2527
```

If the interface you want to configure is a hot-pluggable type, such as a USB or PC Card device, you can see the kernel messages by viewing the */var/log/messages* file; for example, by running `tail -f` on the file before you plug in the device.

3. In addition, it is possible to look up man pages on the Web. Check *http://www.openbsd.org/* and the other project websites. They offer keyword-based man page searching.

Next, you need to configure the interface, first to enable the link, and finally to configure the system for TCP/IP. You can do this from the command line, like this:

```
$ sudo ifconfig ral0 up mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
```

This command does several things at once. It configures the `ral0` interface, enables the interface with the `up` parameter, and specifies that the interface is an access point for a wireless network with `mediaopt hostap`. Then it explicitly sets the operating mode to `11g` and the channel to `11`. Finally, it uses the `nwid` parameter to set the network name to `unwiredbsd`, with the WEP key (`nwkey`) set to the hexadecimal string `0x1deadbeef9`.

Use `ifconfig` to check that the command successfully configured the interface:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect mode 11g hostap
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 nwkey <not displayed> 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Note the contents of the `media` and `ieee80211` lines. The information displayed here should match what you entered on the `ifconfig` command line.

With the link part of your wireless network operational, you can assign an IP address to the interface:

```
$ sudo ifconfig ral0 10.50.90.1
```

On OpenBSD, you can combine both steps into one by creating a */etc/hostname.ral0* file roughly like this:

```
up mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
inet 10.50.90.1
```

Then run **`sh /etc/netstart ral0`** (as root), or wait patiently for your next boot to complete.

Notice that the preceding configuration is divided over two lines. The first line generates an `ifconfig` command that sets up the interface with the correct parameters for the physical wireless network. The second line generates the command that sets the IP address after the first command completes. Because this is our access point, we set the channel explicitly, and we enable weak WEP encryption by setting the `nwkey` parameter.

On NetBSD, you can normally combine all of these parameters in one *rc.conf* setting:

```
ifconfig_ral0="mediaopt hostap mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef inet 10.50.90.1"
```

FreeBSD 8 and newer take a slightly different approach, tying wireless network devices to the unified `wlan(4)` driver. Depending on your kernel configuration, you may need to add the relevant module load lines to */boot/loader.conf*. On one of my test systems, */boot/loader.conf* looks like this:

```
if_rum_load="YES"
wlan_scan_ap_load="YES"
wlan_scan_sta_load="YES"
wlan_wep_load="YES"
wlan_ccmp_load="YES"
wlan_tkip_load="YES"
```

With the relevant modules loaded, setup is a multicommand affair, best handled by a *start_if.if* file for your wireless network. Here is an example of an */etc/start_if.rum0* file for a WEP access point on FreeBSD 8:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 wlanmode hostap"
ifconfig_wlan0="inet 10.50.90.1 netmask 255.255.255.0 ssid unwiredbsd \
wepmode on wepkey 0x1deadbeef9 mode 11g"
```

After a successful configuration, your `ifconfig` output should show both the physical interface and the `wlan` interface up and running:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.1 netmask 0xffffff00 broadcast 10.50.90.255
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
        ssid unwiredbsd channel 6 (2437 Mhz 11g) bssid 00:24:1d:9a:bf:67
        country US authmode OPEN privacy ON deftxkey UNDEF wepkey 1:40-bit
        txpower 0 scanvalid 60 protmode CTS dtimperiod 1 -dfs
```

The line `status: running` means that you are up and running, at least on the link level.

NOTE *Be sure to check the most up-to-date `ifconfig` man page for other options that may be more appropriate for your configuration.*

An OpenBSD WPA Access Point

WPA support was introduced in OpenBSD 4.4, with extensions to most wireless network drivers and a new utility called `wpa-psk(8)` to handle WPA keying operations.

NOTE *There may still be wireless network drivers that do not have WPA support, so check the driver's man page to see if WPA is supported before you try to configure your network to use it. You can combine 802.1x key management with an external authentication server for "enterprise" mode, but we will stick to the simpler preshared key setup for our purposes.*

The procedure for setting up an access point with WPA is quite similar to the one we followed for WEP. To generate a shared WPA key, you need to run the `wpa-psk` utility. If we reuse the WEP key from the earlier examples as the cleartext for our WPA passphrase, we could generate our key like this:

```
$ wpa-psk unwiredbsd 0x1deadbeef9
0x31db31f2291f1ddf3ded3ca463a7dd5c0cd77a814f1d8e6c64990bfcb287b202
```

You could copy this value into the `ifconfig` command or *hostname.if* file, or make `ifconfig` read the output of the `wpa-psk` call directly. Putting the cleartext into the configuration file will also make it slightly more readable. For a WPA setup with a preshared key (sometimes referred to as a *network password*), you would typically write a *hostname.if* file like this:

```
up media autoselect mediaopt hostap mode 11g chan 1 nwid unwiredbsd \
        wpa wpapsk `wpa-psk unwiredbsd 0x1deadbeef9`
inet 10.50.90.1
```

If you are already running the WEP setup described earlier, disable those settings with the following:

```
$ sudo ifconfig ral0 -nwid -nwkey
```

Then enable the new settings with this command:

```
$ sudo sh /etc/netstart ral0
```

You can then check that the access point is up and running with `ifconfig`:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
```

```
        media: IEEE802.11 autoselect mode 11g hostap
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 wpapsk <not displayed>
wpaprotos wpa1,wpa2 wpaakms psk wpaciphers tkip,ccmp wpagroupcipher tkip 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
        inet 10.50.90.1 netmask 0xff000000 broadcast 10.255.255.255
```

Note the `status: active` indication and that the WPA options we did not set explicitly are shown with their sensible default values.

A FreeBSD WPA Access Point

Moving from the WEP access point we configured earlier to a somewhat safer WPA setup is rather straightforward. WPA support on FreeBSD comes in the form of `hostapd` (a program that is somewhat similar to OpenBSD's `hostapd` but not identical). We start by editing the */etc/start_if.rum0* file to remove the authentication information. The edited file should look something like this:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 wlanmode hostap"
ifconfig_wlan0="inet 10.50.90.1 netmask 255.255.255.0 ssid unwiredbsd mode 11g"
```

Next, we add the enable line for `hostapd` in */etc/rc.conf*:

```
hostapd_enable="YES"
```

And finally, `hostapd` will need some configuration of its own, in */etc/hostapd.conf*:

```
interface=wlan0
debug=1
ctrl_interface=/var/run/hostapd
ctrl_interface_group=wheel
ssid=unwiredbsd
wpa=1
wpa_passphrase=0x1deadbeef9
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP TKIP
```

Here, the interface specification is rather self-explanatory, while the `debug` value is set to produce minimal messages. The range is 0 through 4, where 0 is no debug messages at all. You should not need to change the `ctrl_interface*` settings unless you are developing `hostapd`. The first of the next five lines sets the network identifier. The subsequent lines enable WPA and set the passphrase. The final two lines specify accepted key-management algorithms and encryption schemes. (For the finer details and updates, see the `hostapd(8)` and `hostapd.conf(5)` man pages.)

After a successful configuration (running **sudo /etc/rc.d/hostapd forcestart** comes to mind), `ifconfig` should produce output about the two interfaces similar to this:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.1 netmask 0xffffff00 broadcast 10.50.90.255
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g <hostap>
        status: running
        ssid unwiredbsd channel 6 (2437 Mhz 11g) bssid 00:24:1d:9a:bf:67
        country US authmode WPA privacy MIXED deftxkey 2 TKIP 2:128-bit
        txpower 0 scanvalid 60 protmode CTS dtimperiod 1 -dfs
```

The line `status: running` means that you are up and running, at least on the link level.

The Access Point's PF Rule Set

With the interfaces configured, it's time to start configuring the access point as a packet-filtering gateway. You can start by copying the basic gateway setup from Chapter 3. Enable gatewaying via the appropriate entries in the access point's *sysctl.conf* or *rc.conf* file, and then copy across the *pf.conf* file. Depending on the parts of the previous chapter that were most useful to you, the *pf.conf* file may look something like this:

```
ext_if = "re0" # macro for external interface - use tun0 or pppoe0 for PPPoE
int_if = "re1" # macro for internal interface
localnet = $int_if:network
client_out = "{ ssh, domain, pop3, auth, nntp, http,\
                https, cvspserver, 2628, 5999, 8000, 8080 }"
udp_services = "{ domain, ntp }"
icmp_types = "{ echoreq, unreach }"
# ext_if IP address could be dynamic, hence ($ext_if)
match out on $ext_if from $localnet nat-to ($ext_if)
block all
pass quick inet proto { tcp, udp } from $localnet to port $udp_services
pass log inet proto icmp icmp-type $icmp_types
pass inet proto tcp from $localnet port $client_out
```

If you are running a PF version equal to OpenBSD 4.6 or earlier, the `match` rule with `nat-to` instead becomes this:

```
nat on $ext_if from $localnet to any -> ($ext_if)
```

The only difference that is strictly necessary for your access point to work is to change the definition of `int_if` to match the wireless interface. In our example, this means the line should now read as follows:

```
int_if = "ral0"  # macro for internal interface
```

More than likely, you will also want to set up `dhcpd` to serve addresses and other relevant network information to clients after they have associated with your access point. Setting up `dhcpd` is fairly straightforward if you read the man pages.

That's all there is to it. This configuration gives you a functional BSD access point, with at least token security (actually more like a Keep Out! sign) via WEP encryption, or a slightly more robust link-level encryption with WPA. If you need to support FTP, copy the `ftp-proxy` configuration from the machine you set up in Chapter 3 and make changes similar to those you made for the rest of the rule set.

Access Points with Three or More Interfaces

If your network design dictates that your access point is also the gateway for a wired local network, or even several wireless networks, you need to make some minor changes to your rule set. Instead of just changing the value of the `int_if` macro, you might want to add another (descriptive) definition for the wireless interface, such as the following:

```
air_if = "ral0"
```

Your wireless interfaces are likely to be on separate subnets, so it might be useful to have a separate rule for each of them to handle the NAT configuration. Here's an example for OpenBSD 4.7 and newer systems:

```
match out on $ext_if from $air_if:network nat-to ($ext_if)
```

And here's one on pre–OpenBSD 4.7 PF versions:

```
nat on $ext_if from $air_if:network to any -> ($ext_if) static-port
```

Depending on your policy, you might also want to adjust your `localnet` definition, or at least include `$air_if` in your `pass` rules where appropriate. And once again, if you need to support FTP, a separate pass with redirection for the wireless network to `ftp-proxy` may be in order.

Handling IPSec, VPN Solutions

You can set up virtual private networks (VPNs), using built-in IPsec tools, OpenSSH, or other tools. However, with the relatively poor security profile of wireless networks in general, you are likely to want to set up some additional security.

The options fall roughly into three categories:

SSH If your VPN is based on SSH tunnels, the baseline rule set already contains all the filtering you need. Your tunneled traffic will be indistinguishable from other SSH traffic to the packet filter.

IPsec with UDP key exchange (IKE/ISAKMP) Several IPsec variants depend critically on key exchange via `proto udp port 500` and use `proto udp port 4500` for NAT Traversal (NAT-T). You need to let this traffic through in order to let the flows become established. Almost all implementations also depend critically on letting ESP protocol traffic (protocol number 50) pass between the hosts with the following:

```
pass proto esp from $source to $target
```

Filtering on IPsec encapsulation interfaces With a properly configured IPsec setup, you can set up PF to filter on the encapsulation interface `enc0` itself with the following:[4]

```
pass on enc0 proto ipencap from $source to $target keep state (if-bound)
```

See Appendix A for references to some of the more useful literature on the subject.

The Client Side

As long as you have BSD clients, setup is extremely easy. The steps involved in connecting a BSD machine to a wireless network are quite similar to the ones we just went through to set up a wireless access point. On OpenBSD, the configuration centers on the *hostname.if* file for the wireless interface. On FreeBSD, the configuration centers on *rc.conf*, but will most likely involve a few other files, depending on your exact configuration.

OpenBSD Setup

Starting with the OpenBSD case, in order to connect to the WEP access point we just configured, your OpenBSD clients need a *hostname.if* (for example, */etc/hostname.ral0*) configuration file with these lines:

```
up media autoselect mode 11g chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
dhcp
```

The first line sets the link-level parameters in more detail than usually required. Only `up` and the `nwid` and `nwkey` parameters are strictly necessary. In almost all cases, the driver will associate with the access point on the appropriate channel and in the best available mode. The second line calls for a DHCP configuration, and in practice, causes the system to run a `dhclient` command to retrieve TCP/IP configuration information.

4. In OpenBSD 4.8 the encapsulation interface became a cloneable interface, and you can configure several separate `enc` interfaces. All `enc` interfaces become members of the `enc` interface group.

If you chose to go with the WPA configuration, the file will look like this instead:

```
up media autoselect mode 11g chan 1 nwid unwiredbsd wpa wpapsk `wpa-psk unwiredbsd 0x1deadbeef9`
dhcp
```

Again, the first line sets the link-level parameters, where the crucial ones are the network selection and encryption parameters `nwid`, `wpa`, and `wpapsk`. You can try omitting the `mode` and `chan` parameters; in almost all cases, the driver will associate with the access point on the appropriate channel and in the best available mode.

If you want to try out the configuration commands from the command line before committing the configuration to your */etc/hostname.if* file, the command to set up a client for the WEP network is as follows:

```
$ sudo ifconfig ral0 up mode 11b chan 1 nwid unwiredbsd nwkey 0x1deadbeef9
```

The `ifconfig` command should complete without any output. You can then use `ifconfig` to check that the interface was successfully configured. The output should look something like this:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect (OFDM54 mode 11g)
        status: active
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 nwkey <not displayed> 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Note that the `ieee80211` line displays the network name and channel, along with a few other parameters. The information displayed here should match what you entered on the `ifconfig` command line.

Here is the command to configure your OpenBSD client to connect to the WPA network:

```
$ sudo ifconfig ral0 nwid unwiredbsd wpa wpapsk `wpa-psk unwiredbsd 0x1deadbeef9`
```

The command should complete without any output. If you use `ifconfig` again to check the interface status, the output will look something like this:

```
$ ifconfig ral0
ral0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:25:9c:72:cf:60
        priority: 4
        groups: wlan
        media: IEEE802.11 autoselect (OFDM54 mode 11g)
        status: active
```

```
        ieee80211: nwid unwiredbsd chan 1 bssid 00:25:9c:72:cf:60 wpapsk <not displayed>
wpaprotos wpa1,wpa2 wpaakms psk wpaciphers tkip,ccmp wpagroupcipher tkip 100dBm
        inet6 fe80::225:9cff:fe72:cf60%ral0 prefixlen 64 scopeid 0x2
```

Check that the `ieee80211:` line displays the correct network name and sensible WPA parameters.

Once you are satisfied that the interface is configured at the link level, use the `dhclient` command to configure the interface for TCP/IP, like this:

```
$ sudo dhclient ral0
```

The `dhclient` command should print a summary of its dialogue with the DHCP server that looks something like this:

```
DHCPREQUEST on ral0 to 255.255.255.255 port 67
DHCPREQUEST on ral0 to 255.255.255.255 port 67
DHCPACK from 10.50.90.1 (00:25:9c:72:cf:60)
bound to 10.50.90.11 -- renewal in 1800 seconds.
```

FreeBSD Setup

On FreeBSD, you may need to do a bit more work than is necessary with OpenBSD. Depending on your kernel configuration, you may need to add the relevant module load lines to */boot/loader.conf*. On one of my test systems, */boot/loader.conf* looks like this:

```
if_rum_load="YES"
wlan_scan_ap_load="YES"
wlan_scan_sta_load="YES"
wlan_wep_load="YES"
wlan_ccmp_load="YES"
wlan_tkip_load="YES"
```

With the relevant modules loaded, you can join the WEP network we configured earlier by issuing the following command:

```
$ sudo ifconfig wlan create wlandev rum0 ssid unwiredbsd wepmode on wepkey 0x1deadbeef9 up
```

Then issue this command:

```
$ sudo dhclient wlan0
```

For a more permanent configuration, create a *start_if.rum0* file (replace *rum0* with the name of the physical interface if it differs) with content like this:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0 ssid unwiredbsd wepmode on wepkey 0x1deadbeef9 up"
ifconfig_wlan0="DHCP"
```

If you want to join the WPA network, you need to set up `wpa_supplicant` and change your network interface settings slightly. For the WPA access point, connect with the following configuration in your *start_if.rum0* file:

```
wlans_rum0="wlan0"
create_args_wlan0="wlandev rum0"
ifconfig_wlan0="WPA"
```

You also need an */etc/wpa_supplicant.conf* file that contains the following:

```
network={
  ssid="unwiredbsd"
  psk="0x1deadbeef9"
}
```

Finally, add a second `ifconfig_wlan0` line in *rc.conf*, to ensure that `dhclient` runs correctly.

```
ifconfig_wlan0="DHCP"
```

Other WPA networks may require additional options. After a successful configuration, the `ifconfig` output should display something like this:

```
rum0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 2290
        ether 00:24:1d:9a:bf:67
        media: IEEE 802.11 Wireless Ethernet autoselect mode 11g
        status: associated
wlan0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:24:1d:9a:bf:67
        inet 10.50.90.16 netmask 0xffffff00 broadcast 10.50.90.255
        media: IEEE 802.11 Wireless Ethernet OFDM/36Mbps mode 11g
        status: associated
        ssid unwiredbsd channel 1 (2412 Mhz 11g) bssid 00:25:9c:72:cf:60
        country US authmode WPA2/802.11i privacy ON deftxkey UNDEF
        TKIP 2:128-bit txpower 0 bmiss 7 scanvalid 450 bgscan bgscanintvl 300
        bgscanidle 250 roam:rssi 7 roam:rate 5 protmode CTS roaming MANUAL
```

Guarding Your Wireless Network with authpf

Security professionals tend to agree that even though WEP encryption offers little protection, it's just barely enough to signal to would-be attackers that you do not intend to let all and sundry use your network resources. Using WPA increases security somewhat, at the cost of some complexity.

The configurations we built so far in this chapter are functional. Both the WEP and WPA configurations will let all reasonably configured wireless clients connect, and that may be a problem in itself, since that configuration does not have any real support built in for letting you decide who uses your network.

As mentioned earlier, MAC address filtering is not really a solid defense against attackers because it's just too easy to change a MAC address. The OpenBSD developers chose a radically different approach to this problem when they introduced `authpf` in OpenBSD version 3.1. Instead of tying access to a hardware identifier such as the network card's MAC address, they decided that the robust and highly flexible *user* authentication mechanisms already in place were more appropriate for the task. The user shell `authpf` lets the system load PF rules on a per-user basis, effectively deciding which user gets to do what.

To use `authpf`, you create users with the `authpf` program as their shell. In order to get network access, the user logs in to the gateway using SSH. Once the user successfully completes SSH authentication, `authpf` loads the rules you have defined for the user or the relevant class of users.

These rules, which apply to the IP address the user logged in from, stay loaded and in force for as long as the user stays logged in via the SSH connection. Once the SSH session is terminated, the rules are unloaded, and in most scenarios, all non-SSH traffic from the user's IP address is denied. With a reasonable setup, only traffic originated by authenticated users will be let through.

NOTE *On OpenBSD, `authpf` is one of the login classes offered by default, as you will notice the next time you create a user with `adduser`.*

For systems where the `authpf` login class is not available by default, you may need to add the following lines to your */etc/login.conf* file:

```
authpf:\
        :welcome=/etc/motd.authpf:\
        :shell=/usr/sbin/authpf:\
        :tc=default:
```

The next couple of sections contain a few examples that may or may not fit your situation directly, but that I hope will give you ideas you can use.

A Basic Authenticating Gateway

Setting up an authenticating gateway with `authpf` involves creating and maintaining a few files besides your basic *pf.conf*. The main addition is *authpf.rules*. The other files are fairly static entities that you will not be spending much time on once they have been created.

Start by creating an empty */etc/authpf/authpf.conf* file. This file needs to be there in order for `authpf` to work, but it doesn't actually need any content, so creating an empty file with `touch` is appropriate.

The other relevant bits of */etc/pf.conf* follow. First, here are the interface macros:

```
ext_if = "re0"
int_if = "ath0"
```

In addition, if you define a table called <authpf_users>, authpf will add the IP addresses of authenticated users to the table:

```
table <authpf_users> persist
```

If you need to run NAT, the rules that take care of the translation could just as easily go in *authpf.rules*, but keeping them in the *pf.conf* file does not hurt in a simple setup like this:

```
pass out on $ext_if from $localnet nat-to ($ext_if)
```

Here's pre–OpenBSD 4.7 syntax:

```
nat on $ext_if from $localnet to any -> ($ext_if)
```

Next, we create the authpf anchor, where rules from *authpf.rules* are loaded once the user authenticates:

```
anchor "authpf/*"
```

For pre–OpenBSD 4.7 authpf versions, several anchors were required, so the corresponding section would be as follows:

```
nat-anchor "authpf/*"
rdr-anchor "authpf/*"
binat-anchor "authpf/*"
anchor "authpf/*"
```

This brings us to the end of the required parts of a *pf.conf* file for an authpf setup.

For the filtering part, we start with the block all default, and then add the pass rules we need. The only essential item at this point is to let SSH traffic pass on the internal network:

```
pass quick on $int_if inet proto { tcp, udp } to $int_if port ssh
```

From here on out, it really is up to you. Do you want to let your clients have name resolution before they authenticate? If so, put the pass rules for the TCP and UDP service domain in your *pf.conf* file, too.

For a relatively simple and egalitarian setup, you could include the rest of our baseline rule set, changing the pass rules to allow traffic from the addresses in the <authpf_users> table, rather than any address in your local network:

```
pass quick inet proto { tcp, udp } from <authpf_users> to port $udp_services
pass inet proto tcp from <authpf_users> to port $client_out
```

For a more differentiated setup, you could put the rest of your rule set in */etc/authpf/authpf.rules* or per-user rules in customized *authpf.rules* files in each user's directory under */etc/authpf/users/*. If your users generally need some protection, your general */etc/authpf/authpf.rules* could have content like this:

```
client_out = "{ ssh, domain, pop3, auth, nntp, http, https }"
udp_services = "{ domain, ntp }"
pass quick inet proto { tcp, udp } from $user_ip to port $udp_services
pass inet proto tcp from $user_ip to port $client_out
```

The macro `user_ip` is built into `authpf` and expands to the IP address from which the user authenticated. These rules will apply to any user who completes authentication at your gateway.

A nice and relatively easy addition to implement is special-case rules for users with different requirements than your general user population. If an *authpf.rules* file exists in the user's directory under */etc/authpf/users/*, the rules in that file will be loaded for the user. This means that your naïve user Peter who only needs to surf the Web and have access to a service that runs on a high port on a specific machine could get what he needs with a */etc/authpf/users/peter/authpf.rules* file like this:

```
client_out = "{ domain, http, https }"
pass inet from $user_ip to 192.168.103.84 port 9000
pass quick inet proto { tcp, udp } from $user_ip to port $client_out
```

On the other hand, Peter's colleague Christina runs OpenBSD and generally knows what she is doing, even if she sometimes generates traffic to and from odd ports. You could give her free rein by putting this in */etc/authpf/users/christina/authpf.rules*:

```
pass from $user_ip os = "OpenBSD" to any
```

This means Christina can do pretty much anything she likes over TCP as long as she authenticates from her OpenBSD machines.

Wide Open but Actually Shut

In some settings, it makes sense to set up your network to be open and unencrypted at the link level, while enforcing some restrictions via `authpf`. The next example is very similar to Wi-Fi zones you may encounter in airports or other public spaces, where anyone can associate to the access points and get an IP address, but any attempt at accessing the Web will be redirected to one specific web page until the user has cleared some sort of authentication.[5]

This *pf.conf* file is again built on our baseline, with two important additions to the basic `authpf` setup: a macro and a redirection.

5. Thanks to Vegard Engen for the idea and showing me his configuration, which is preserved here in spirit, if not all details.

```
ext_if = "re0"
int_if = "ath0"
auth_web="192.168.27.20"
dhcp_services = "{ bootps, bootpc }" # DHCP server + client
table <authpf_users> persist
pass in quick on $int_if proto tcp from ! <authpf_users> to port http rdr-to $auth_web
match out on $ext_if from $int_if:network nat-to ($ext_if)
anchor "authpf/*"
block all
pass quick on $int_if inet proto { tcp, udp } to $int_if port $dhcp_services
pass quick inet proto { tcp, udp } from $int_if:network to any port domain
pass quick on $int_if inet proto { tcp, udp } to $int_if port ssh
```

For older `authpf` versions, use this file instead:

```
ext_if = "re0"
int_if = "ath0"
auth_web="192.168.27.20"
dhcp_services = "{ bootps, bootpc }" # DHCP server + client
table <authpf_users> persist
rdr pass on $int_if proto tcp from ! <authpf_users> to any port http -> $auth_web
nat on $ext_if from $localnet to any -> ($ext_if)
nat-anchor "authpf/*"
rdr-anchor "authpf/*"
binat-anchor "authpf/*"
anchor "authpf/*"
block all
pass quick on $int_if inet proto { tcp, udp } to $int_if port $dhcp_services
pass quick inet proto { tcp, udp } from $int_if:network to port domain
pass quick on $int_if inet proto { tcp, udp } to $int_if port ssh
```

The `auth_web` macro and the redirection make sure all web traffic from addresses that are not in the `<authpf_users>` table leads all nonauthenticated users to a specific address. At that address, you set up a web server that serves up whatever you need. This could range from a single page with instructions on who to contact in order to get access to the network all the way up to a system that accepts credit cards and handles user creation.

Note that in this setup, name resolution will work, but all surfing attempts will end up at the `auth_web` address. Once the users clear authentication, you can add general rules or user-specific ones to the *authpf.rules* files as appropriate for your situation.