# 4

# WIRELESS NETWORKS MADE EASY

It is rather tempting to say that on BSD, and on OpenBSD in particular, there's no need to "make wireless networking simple," because it already is. Getting a wireless network running is not very different from getting a wired one running, but there are some issues that turn up simply because we are dealing with radio waves and not wires. We will take some time to 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 will 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 is, in principle, a two-step process: First, establish a link, and then 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 right kind and probably negotiate some additional parameters.

Fortunately, on BSD systems all configuration of wireless network devices happens via `ifconfig` commands and options, just as you would set up any other network interface.[1]

Still, since we are introducing wireless networks here, we need to look at the 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 in the following sections.

**NOTE** *For a more complete overview of issues surrounding security in wireless networks see, for example, Professor Kjell Jørgen Hole's articles and slides at his site* (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 at* http://wifinetnews.com/archives/cat_security.html *and "The Unofficial 802.11 Security Web Page" at* http://www.drizzle.com/~aboba/IEEE *come 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 is possible to change the MAC address of essentially any wireless network adapter on the market today.[2]

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. We will be looking at bridges and some of the more useful ways to use them with packet filtering in the next chapter.

---

[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 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`.

## *WEP*

One consequence of using radio waves instead of wires to move data is that it is comparatively easy for outsiders to capture your data in transit. 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, which 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 home brew 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 it is still widely supported and used. Essentially all IEEE 802.11 equipment out there has support for at least WEP, and a surprising number of products offer MAC address filtering, too.

You should consider network traffic protected only by WEP to be only 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 arguably too complicated for widespread implementation. In addition, WPA has also 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. If your project specification includes WPA, look carefully at your operating system and driver documentation.

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.

## *Picking the Right Hardware for the Task*

Picking the right hardware is not necessarily a daunting task. On a BSD system, one simple

```
$ apropos wireless
```

command is all you need to enter to see a listing of all manual pages with the word *wireless* in their subject lines.[3]

Even on a freshly installed system, this 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 testing 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.

## Setting Up a Simple Wireless Network

To start building 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 we introduce over the next few paragraphs. After all, it is more convenient than starting a new configuration from scratch.

**NOTE** *We are in infrastructure-building mode here, and we will be setting up the access point first. If you prefer to look at the client side first, see "The Client Side" on page 40, and then come back to this page.*

As we mentioned earlier, 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 also get stored in the file */var/run/dmesg.boot*. You can view the file itself or use the output of the `dmesg` command. With a successfully configured PCI card, you should see something like this:

```
ath0 at pci1 dev 4 function 0 "Atheros AR5212" rev 0x01: irq 11
ath0: AR5212 5.6 phy 4.1 rf5111 1.7 rf2111 2.3, ETSI1W, address 00:0d:88:c8:a7:c4
```

If the interface you want to configure is a hot-pluggable type such as a USB or PCCARD 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.

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

```
$ sudo ifconfig ath0 up mediaopt hostap mode 11b chan 11 nwid unwiredbsd nwkey 0x1deadbeef9
```

[3] In addition, it is possible to look up man pages on the Web. Check *http://www.openbsd.org* and the other projects' websites; they offer keyword-based man page searches.

This command does several things at once. It configures the `ath0` 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 11b, explicitly sets the channel to 11, and finally, 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 ath0
ath0: flags=8823<UP,BROADCAST,NOTRAILERS,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:11:95:ca:e6:59
        groups: wlan
        media: IEEE802.11 autoselect mode 11b hostap
        status: no network
        ieee80211: nwid unwiredbsd chan 11 bssid 00:11:95:ca:e6:59 nwkey <not displayed>
        inet6 fe80::211:95ff:feca:e659%ath0 prefixlen 64 tentative scopeid 0x5
```

Note the contents of the `media:` and `ieee80211:` lines. They should match what you entered on the `ifconfig` command line. With the link part of your wireless network operational, you can go on to the next step and assign an IP address to the interface:

```
sudo ifconfig ath0 10.50.90.1
```

On OpenBSD, you can achieve both by creating a */etc/hostname.ath0* file, roughly like this:

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

and either running `/etc/netstart ath0` (you need to do that as root) or waiting patiently for your next boot to complete.

Note that the 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 command, which sets the IP address, is executed only after the first one completes. It is worth noting that since this is our access point, we set the channel explicitly, and we enable a weak WEP encryption by setting the `nwkey` parameter.

On FreeBSD and NetBSD, you can normally combine all the parameters in one *rc.conf* setting:

```
ifconfig_ath0="mediaopt hostap mode 11b chan 11 nwid unwiredbsd nwkey 0x1deadbeef inet 10.50.90.1"
```

However, on some hardware combinations, setting the link-level options and the IP address at the same time fails. If your one-liner configuration fails, you will need to put the two lines in your */etc/start_if.ath0* and substitute your interface name for *ath0* if required.

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

### 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; then copy across the *pf.conf* file. Depending on the parts of the last 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)
nat on $ext_if from $localnet to any -> ($ext_if)
block all
pass quick inet proto { tcp, udp } from $localnet to any port $udp_services
pass log inet proto icmp all icmp-type $icmp_types
pass inet proto tcp from $localnet to any port $client_out
```

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

```
int_if = "ath0"  # macro for internal interface
```

More likely than not, 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. If you need to support FTP, you can copy the ftp-proxy configuration from the machine you set up in Chapter 3 and make similar changes as for the rest of the rule set.

### If Your Access Point Has 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

```
air_if = "ath0"
```

In a wireless gateway configuration, your wireless interfaces are likely to be on separate subnets, so it might be useful for each of them to have its own `nat` rule as well:

```
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 redirection for the wireless network to ftp-proxy may be in order.

## *Handling IPsec, VPN Solutions*

The details of setting up Virtual Private Networks (VPNs) using the built-in IPsec tools, OpenSSH, or other tools are beyond the scope of this chapter. However, with the relatively poor security profile of wireless networks in general, you are likely to want to set up some additional security. A VPN setup may range from useful to essential in your situation.

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 `proto {tcp, udp} port 4500` for NAT Traversal (NAT-T). You need to let this traffic through in order to let the flows become established. Some implementations also depend critically on letting ESP protocol traffic (protocol 50) pass between the hosts: `pass proto esp from $source to $target`.

**Filtering on the IPsec encapsulation interface**
With a properly configured IPsec setup, you can set up PF to filter on the encapsulation interface `enc0` itself: `pass on enc0 proto ipencap from $source to $target keep state (if-bound)`.

Check Appendix A for references to some useful literature on the subject.

### *The Client Side*

As long as you have all BSD clients, setup is extremely easy. In order to connect to the access point we just configured, your OpenBSD clients would need a *hostname.if* configuration file with

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

Try these out from the command line first, with

```
$ sudo ifconfig ath0 up mode 11b chan 11 nwid unwiredbsd nwkey 0x1deadbeef9
```

followed by

```
$ sudo dhclient ath0
```

The `ifconfig` command should complete without any output, while the `dhclient` command should print a summary of its dialog with the DHCP server:

```
DHCPREQUEST on ath0 to 255.255.255.255 port 67
DHCPREQUEST on ath0 to 255.255.255.255 port 67
DHCPACK from  10.50.90.1
bound to 10.50.90.11 -- renewal in 1800 seconds.
```

Again on FreeBSD, you would need to put those lines in your */etc/start_if.ath0* and substitute your interface name for *ath0* if required.

## Guarding Your Wireless Network with authpf

As always, there are other ways to configure the security of your wireless network besides the one we have just seen. What little protection WEP encryption offers, security professionals tend to agree, is barely enough to signal to an attacker that you do not intend to let all and sundry use your network resources.

The configuration we built in “Setting Up a Simple Wireless Network” on page 36 is functional. It 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 we mentioned earlier, MAC address filtering is not really a solid defense against attackers. Changing the MAC address is just too easy. 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, the OpenBSD developers decided that the robust and highly flexible *user* authentication mechanisms already in place were more appropriate for the task. The authpf tool is a user shell that 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.

It is worth noting that on OpenBSD, authpf is one of the login classes that is offered by default, as you will notice the next time you create a user with the adduser program.

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

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

The following sections contain a few examples that may or may not fit your situation directly, but I hope they 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 with creating an empty */etc/authpf/authpf.conf*. It needs to be there 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, we create the interface macros:

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

In addition, authpf requires a table to fill with the IP addresses of authenticated users:

```
table <authpf_users> persist
```

The nat rules, if you need them, 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:

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

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

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

That brings us to the end of the required parts of a *pf.conf* 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 thing we really need at this point is to pass `ssh` 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,* too.

For a relatively simple and egalitarian setup, you could include the rest of our baseline rule set, but change 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 any port $udp_services
pass inet proto tcp from <authpf_users> to any port $client_out
```

For a more differentiated setup, you could put the rest of your rule set in */etc/authpf/authpf.rules* or put per-user rules in customized *authpf.rules* files in each user's directory under */etc/authpf/users/*. If your users normally 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 any port $udp_services
pass inet proto tcp from $user_ip to any port $client_out
```

The macro `$user_ip` is built into authpf and expands to the IP address the user authenticated from. These rules 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 Windows 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 any port $client_out
```

On the other hand, Peter's colleague Christina runs OpenBSD and generally knows what she's doing, even if she sometimes generates traffic to and from odd ports. You could let her have 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 situations 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 can encounter in airports or other public spaces, in which 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.[4]

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

```
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/*"
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
```

[4] Thanks to Vegard Engen for the idea and showing me his configuration, which is preserved here in spirit, if not in every detail.

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 webserver that serves up whatever it is you need. This could be anything 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 account creation.

It is worth noting that name resolution will work in this setup, 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.