Exploring the Mac OS X Firewall
by Peter Hickman
03/15/2005

All Unix systems have a firewall. But on the Mac, it's installed by default and you can't turn it off. To keep things simple, and keep users from having to build their own firewall rules, OS X packages the firewall as a sub-menu of Sharing within System Preferences. The benefit of this approach is that the firewall can be tied to services. Turn on Personal Web Sharing, for example, and a new rule appears in the firewall to control access to the service.

Presently, I have Personal Web Sharing turned on in the Services menu. The firewall is turned on. And in the Firewall menu, there's a checkbox to allow access to my web server. Opening a terminal and typing sudo ipfw list allows me to see the actual firewall rules that Mac OS X has created.

```
02000 allow ip from any to any via lo*
02010 deny ip from 127.0.0.0/8 to any in
02020 deny ip from any to 127.0.0.0/8 in
02030 deny ip from 224.0.0.0/3 to any in
02040 deny tcp from any to 224.0.0.0/3 in
02050 allow tcp from any to any out
02060 allow tcp from any to any established
02070 allow tcp from any to any 80 in
02080 allow tcp from any to any 427 in
12190 deny tcp from any to any
65535 allow ip from any to any
```

ipfw is the heart of the Macintosh firewall; the listing is the set of rules that Mac OS X built based on the services I wanted to run. Apache runs on port 80 (for http), while port 427 is the service locator protocol. The first column is the rule number, and the rest is the rule itself. I will go into greater detail later. But for the moment, I will turn on Remote Login. A quick glance at the Firewall menu reveals that the Remote Login rules have been turned on. Looking at the output of sudo ipfw list shows how the rules were changed to allow the new service to run.

```
02000 allow ip from any to any via lo*
02010 deny ip from 127.0.0.0/8 to any in
02020 deny ip from any to 127.0.0.0/8 in
02030 deny ip from 224.0.0.0/3 to any in
02040 deny tcp from any to 224.0.0.0/3 in
02050 allow tcp from any to any out
02060 allow tcp from any to any established
02070 allow tcp from any to any 22 in
02080 allow tcp from any to any 80 in
02090 allow tcp from any to any 427 in
12190 deny tcp from any to any
65535 allow ip from any to any
```

The rules have been rewritten, and a new rule has been added to handle Remote Login. Remote Login is handled by ssh, which operates over port 22. I will turn the firewall off as a final test before I put everything back the way it was.

```
65535 allow ip from any to any
```

So even when we turn the firewall off, it is still running. It's just the rules that have changed. What I'm going to do in this article is explore ipfw to see how I can use it to better secure my Macintosh and learn something of the threats that assail our computers every day. But first, a few words of caution.

Don't Play with Fire

The firewall that comes with the Mac is a good and robust tool and will protect you from many things, and you should have it turned on all the time. Just because we can handcraft rules does not mean that we should. If we get it wrong, we could stop our computer from functioning and create security holes. In this article, we're going to play with the firewall, learn a few things about our Mac, and then just let the System Preferences take care of things (in most cases).

This is one area where a little knowledge can be dangerous. And it's important to remember that after reading this article you will not be an expert on firewalls.

Say Hello to ipfw

To get a feel for what your firewall is having to deal with, you'll get it to log all the decisions that it makes over the course of a few hours. This will show you what's really happening. First, create a file that contains a copy of the rules you're running. Call this file rules.

```
1.  #!/bin/sh
2.
3.  IPFW='/sbin/ipfw -q'
4.  $IPFW -f flush
5.  $IPFW add 2000 allow ip from any to any via lo*
6.  $IPFW add 2010 deny log ip from 127.0.0.0/8 to any in
7.  $IPFW add 2020 deny log ip from any to 127.0.0.0/8 in
8.  $IPFW add 2030 deny log ip from 224.0.0.0/3 to any in
9.  $IPFW add 2040 deny log tcp from any to 224.0.0.0/3 in
10. $IPFW add 2050 allow log tcp from any to any out
11. $IPFW add 2060 allow log tcp from any to any established
12. $IPFW add 2070 allow log tcp from any to any 22 in
13. $IPFW add 2080 allow log tcp from any to any 80 in
14. $IPFW add 2090 allow log tcp from any to any 427 in
15. $IPFW add 12190 deny log tcp from any to any
```

These rules are a copy of the rules that Mac OS X generated for Personal Web Sharing and Remote Login with logging turned on. You do not need to enter rule 65535 as this is hard-coded into the firewall and cannot be changed. To run the rules just type in sudo sh rules followed by sudo ipfw list to make sure that they have loaded correctly. The final step is to turn logging on. Despite the rules requiring that the output be logged, this only actually happens when it is turned on with sysctl.

```
sudo sysctl -w net.inet.ip.fw.verbose=1
```

This might seem complicated, but it allows us to have logging permanently turned on in our rules, only writing to the log file when we want to. The logging is being written to /var/log/system.log. Logging can then be turned on and off without changing or even reloading the rules. To turn logging off, change the 1 to a 0. Leave this running for a few hours as we do all sorts of things that may generate traffic over the Internet. Just one little note of caution at this point: There will be one line in the log file for each packet processed by the firewall. The log file can get very large very quickly! Remember to come back and turn the logging off before your disk fills up.

Looking at the Log Results

/var/log/system.log now contains a large number of lines—one per packet that was handled by the firewall rules we created. Here is an example:

Nov 9 21:12:18 Peter-Hickmans-Computer kernel: ipfw: 2060
Accept TCP 216.65.98.71:119 192.168.1.100:54609 in via en0

After all the date and time cruft, we come to the action that was recorded, which says that rule number 2060 accepted an inbound TCP connection via the primary Ethernet interface from address 216.65.98.71 on port 119 to my computer (on address 192.168.1.100) port 54609. Port 119 is used to connect to Usenet news servers, and port 54609 was just a random port Mac OS X allocated for my program to make the connection. To make any sense of the 341,310 lines that the firewall logged while I had it turned on, we need a program to summarize the log. Let's call this program **checkfw.pl**.

```
1.  #!/usr/bin/perl -w
2.
3.  use strict;
4.  use warnings;
5.
6.  my %connections;
7.
8.  while ( my $line = <> ) {
9.    next unless $line =~ m/ ipfw: /;
10.
11.   $line =~ s/.* ipfw: //g;
12.
13.   my ( $action, $from, $to, $direction )
14.     = ( split( ' ', $line, 7 ) )[ 1, 3, 4, 5 ];
15.
16.   if ( check_address($from) and check_address($to) ) {
17.     my $key = ( $direction eq 'out' ) ?
18.       "$from $to" : "$to $from";
19.     $connections{$action}->{$key}->{$direction}++;
20.   }
21. }
22.
23. foreach my $action ( sort keys %connections ) {
24.   report( $action, %{ $connections{$action} } );
25. }
26.
27. sub report {
28.   my ( $action, %data ) = @_;
29.
30.   print "$action\n";
31.   printf( "%21s Dir %21s : %8s : %8s\n",
32.     'Inside IP', 'Outside IP', 'In', 'Out' );
33.   print '-' x 69 . "\n";
34.
35.   foreach my $k ( sort keys %data ) {
36.     my ( $inside, $outside ) = split( ' ', $k );
37.
38.     my $direction = '-->';
39.     if ( $data{$k}->{in} ) {
40.       $direction = ( $data{$k}->{out} ) ? '<->' : '<--';
41.     }
42.     printf( "%21s %s %21s : %8d : %8d\n",
43.       $inside,
44.       $direction,
45.       $outside,
46.       $data{$k}->{in} || 0,
47.       $data{$k}->{out} || 0 );
48.   }
```

```
49.   print "\n";
50. }
51.
52. ##
53. ## Filter the broken lines in the log
54. ##
55.
56. sub check_address {
57.   my $text = shift;
58.
59.   # There should only be 1 colon
60.   my $count = $text =~ tr/://;
61.   return undef if $count != 1;
62.
63.   # There should only be three .s
64.   $count = $text =~ tr/.//;
65.   return undef if $count != 3;
66.
67.   my ( $ip, $port ) = split( ':', $text );
68.
69.   # Is the port number sensible
70.   return undef if $port < 1 or $port > 65535;
71.
72.   # Are the address digits sensible
73.   foreach my $x ( split( '\.', $ip ) ) {
74.     return undef if $x < 0 or $x > 255;
75.   }
76.
77.   # All is fine
78.   return 1;
79. }
```

When run from the command line with the system.log file for input, it reads all the lines related to the firewall and extracts the action (Accept or Deny), from address, to address, and direction from the rule, then builds up a hash to summarize the data. The rules are always expressed as "from address" followed by "to address," so we need to switch them around so that packets going from B to A inbound are correctly paired with lines for A to B outbound. The check_address function is to weed out the few lines that logging seems to screw up every now and then. Here's an edited sample of the output. Let's first look at the traffic we accepted:

```
Accept
Inside IP Dir           Outside IP :      In :        Out
-------------------------------------------------------------
192.168.1.100:22    <->  192.168.1.1:1059 :    80 :       81
192.168.1.100:45632 <->  66.98.246.15:80 :      1 :        1
192.168.1.100:54609 <->  216.65.98.71:119 : 19768 :    13114
192.168.1.100:54788 <->  216.65.98.71:119 :    15 :       17
192.168.1.100:54789 <->  216.65.98.71:119 :    15 :       14
192.168.1.100:54790 <->  216.65.98.71:119 : 24924 :    16792
192.168.1.100:54799 <->  62.253.162.50:110 :   89 :       82
192.168.1.100:80    <->  210.246.12.35:10611 : 11 :        6
192.168.1.100:80    <->  210.246.12.35:11823 :   11 :      10
```

First, an assumption: Any port number that is four or less digits long is a legitimate port. If it is five digits long, then it was just a port allocated to the process that initiated the connection. This is not always the case but it is a very useful rule of thumb.

The first line is a Remote Login connection (ssh on port 22) to my computer, and the next is a connection to a web site (http on port 80) from my computer. The next four are Usenet connections (nntp on port 119) from my computer followed by a connection from my computer to my ISP's mail server (pop3 on port 110). The next two are connections to the web server on my computer by another computer.

If you see a port number and are not sure what it is used by, look in /etc/services where most of the legitimate port numbers are listed. If you can't find it there or feel that you are not using the service, then Google for "port xxxx". Now let's look at the traffic we rejected:

Deny

```
Inside IP Dir          Outside IP :          In :    Out
-----------------------------------------------------------------
192.168.1.100:1023 <--   218.23.26.94:2385 :    1 :      0
192.168.1.100:1025 <--    80.13.205.45:3369 :    1 :      0
192.168.1.100:1080 <--    67.150.225.61:80 :     1 :      0
192.168.1.100:21   <--   172.210.114.241:4535 :  2 :      0
192.168.1.100:25   <--   222.156.17.50:2304 :    2 :      0
192.168.1.100:2745 <--    80.108.169.225:1248 :  3 :      0
192.168.1.100:3127 <--    24.48.194.98:4609 :    3 :      0
192.168.1.100:4000 <--   159.226.150.135:1505 :  2 :      0
192.168.1.100:4899 <--    80.200.148.8:4937 :    1 :      0
192.168.1.100:5554 <--   200.28.99.6:4257 :      1 :      0
192.168.1.100:6112 <--   211.138.113.23:57949 :  2 :      0
192.168.1.100:9898 <--   218.16.83.117:3700 :    1 :      0
```

As these are all denied packets we are only interested in the port number that they were trying to connect to. Let's go over them:

1023

The Sasser worm runs an FTP server on port 1023 of infected machines. 218.23.26.94 is scanning for unpatched Windows machines still infected by the Sasser worm.

1025

Officially the port used by blackjack servers but also used by a backdoor trojan called PWSteal.ABCHlp. Again, I am being scanned by a computer looking for a backdoor.

1080

Officially used by the SOCKS proxy server. 67.150.225.61 is scanning me, hoping to find a proxy server that it can use to cover its tracks with.

21

At last a legitimate port, FTP. Here I am being scanned to see if I am running an FTP server that can be hacked by someone wanting somewhere to store some files or a place from which to run a warez site. I don't run an FTP server.

25

Another legitimate port, this time SMTP. This is someone looking for an open mail server to send spam through.

2745

Officially the port used by the urbisnet service but more commonly a backdoor installed by the W32.Beagle.E@mm worm.

You get the picture. It just goes on and on. Despite being immune to all these WinTel worms and viruses, we should note that there are people out there who will notice very quickly if we turn on our own

mail or FTP servers. The same scripts that probe your computer for these services are designed to exploit them the moment they get a positive result. This is why we run a firewall and why we don't run unnecessary services.

Writing the ipfw Log Lines to a Different File

The system.log file is a general dumping ground for all log messages, but it has the facility to write specific messages to separate files. In this case, we're going to write the log lines from ipfw into /var/log/ipfw.log. This file already exists on my system, but at the moment, the /etc/syslogd.conf is not set up to send the messages to it. First, I must edit the /etc/syslogd.conf and add the following to the beginning of the file:

1. # Exclude ipfw from the main log
2. !-ipfw
3. This stops the ipfw log lines from being written to the system.log. Now add the following to the end of the file to get the ipfw log lines written to the correct file:
4. # Log ipfw to its own log
5. !ipfw
6. *.* /var/log/ipfw.log

Now it's time to restart syslogd so that the new configuration is loaded. There are two ways of doing this: the Windows way or the Unix way. Rebooting your computer is the Windows way, so we'll do it the Unix way.

```
sudo kill -s HUP `cat /var/run/syslog.pid`
```

The preceding line reads the pid of the existing syslogd process and gets it to reload its configuration file. All that remains is to turn logging on, sudo sysctl -w net.inet.ip.fw.verbose=1, generate some traffic, and look at the contents of the /var/log/ipfw.log file. No lines are being written to the system.log file any more. The main reason for this is that the ipfw logging generates so many lines in the system.log file that it makes finding other, non ipfw, messages much harder.

Writing Your Own Rules

Each packet that wants to pass through the firewall, in or out, is checked against the rules in the order in which they are numbered. Once a packet matches a rule, the assigned action is taken, and the next packet is processed. This is why rule 65535 is hard-coded into the firewall; it will match anything that does not match any other rule. Let's walk through the rules we created earlier.

```
02000 allow ip from any to any via lo*`
```

The first rule allows any ip packet to any address via the loopback interface. The computer makes use of the loopback interface to talk to itself. Trust me when I say that this is useful.

```
02010 deny ip from 127.0.0.0/8 to any in
02020 deny ip from any to 127.0.0.0/8 in
02030 deny ip from 224.0.0.0/3 to any in
02040 deny tcp from any to 224.0.0.0/3 in
```

These rules, and others like them, are there to stop packets with spoofed IP addresses from entering. A more comprehensive set of rules would ban all private IP ranges. The anti-spoofing rules are placed here to stop any packets before they get to the rules that might let them in. If we wanted to bar known bad addresses, this would be the place to do it.

```
02050 allow tcp from any to any out
02060 allow tcp from any to any established
```

Here we allow any outbound packets through and follow this up by allowing any previously established connections back in. The firewall is "state-full"—that is to say it doesn't just process a packet and forget about it as it moves onto the next one. It remembers that it allowed a connection from my computer to my ISP's mail server and therefore can identify incoming packets as being part of the same connection and allow then back in without a whole host of new rules.

```
02070 allow tcp from any to any 22 in
02080 allow tcp from any to any 80 in
02090 allow tcp from any to any 427 in
```

This rule allows any inbound packet to the ports of the services that we are running in. This leaves just one more rule.

12190 deny tcp from any to any

This rule denies any tcp packet that has gotten this far.

Let's review. We have thrown out the spoofed addresses, we have allowed all outbound packets and all established inbound connections, and we have allowed inbound packets to the ports of the services we are running. All that is left are inbound packets to ports we have not allowed. It's a good thing we have stopped them here as rule 65535 would have let them in!

A quick overview of the rule set:

- Allow loopback.
- Deny spoofing and people we don't like.
- Allow outbound connections.
- Allow established connections.
- Allow specific inbound connections.
- Deny everything else.

All rules have the same basic structure, and a lot can be achieved with the simplest of rules. The basic shape of a rule, more information on which can be found with man ipfw, is quite simple:

[number] action [log] proto from src to dst [interface-spec]

If a rule is given without a number, the next number in sequence is taken. If a number is given, the rule will be positioned among the existing rules. If a rule with that number already exists, then the new rule will be added after the old one, and you will have two rules with the same number. The action can take many values, but for now we're only interested in allow and deny.

Allow will allow any matching packet to pass through the firewall, and deny will just drop them onto the floor.

The optional log parameter will write a line to the logfile for each packet that matched the rule. Lines are only written to the logfile if the system variable net.inet.ip.fw.verbose is set to 1. So we don't have to change the rules when we want to turn logging on and off.

The protocol is the type of packet. The only ones we are concerned with are tcp or ip (which is another way of saying all of them). As a rule of thumb all the protocols are layered over ip, and the most common one we encounter is tcp, with icmp and udp after that.

If you look in /etc/protocols you will see that there are a large number of protocols, most of which we will never encounter outside of this file. Most traffic is tcp, and our rules concentrate on this unless we are blocking something wholesale, in which case we use ip to mean any protocol. But a point of caution: In our "Deny everything else" rule, 12190, we only deny the tcp packets; changing this to deny ip packets may stop your computer from working as many behind-the-scene services still need to get through, such as DNS.

The src and dst are the source and destination addresses that can be defined as either:

any - any ip address
a specific ip address, 207.68.172.246, or hostname, msn.com
a netmask, 207.68.128.0/18 or 1.2.3.4:255.255.240.0

The address can be prefixed by not. Additionally, the address can be followed by a list of port numbers or service names (22 or ssh). For example, the following rules allow FTP connections:

allow tcp from any to any 20-21 in
allow tcp from any 20,21 to any 1024-65535 in

The two rules read:

Allow in any inbound tcp packets to ports 20 and 21 (ftp-data and ftp).
Allow in any inbound tcp packets coming from ports 20 or 21 that are trying to connect to ports 1024 to 65536.

The final element of the rule is the interface-spec. For simple rules, in and out are sufficient and, if omitted, the default is to allow both in and out. When added to a rule they limit the action of the rule to check only the inbound or outbound packets. The interface-spec allows much more control than we are showing here, for example, a rule about being bound to particular interfaces such as an Airport card rather then the normal RJ45 socket.

If we run ifconfig -l, we will get a list of interfaces that are available on our computer. Mine lists six, even though only three are active. The rules we are using will apply to any packet passing through any valid interface.

Our Own Startup Script

The statement "It's not possible to override the firewall built into Mac OS X" isn't entirely true. We could go in and hack around with the kext files, but this is more work than we really need to undertake. What we can do is create our own startup script that runs after the existing firewall to implement our rules. We need to create a directory called /Library/StartupItems/Firewall and include in it two files. The first is a generic startup script called Firewall.

```
#!/bin/sh

##
# Firewall
##

. /etc/rc.common

StartService ()
{
  if [ "${FIREWALL:=-NO-}" = "-YES-" ]
  then
    ConsoleMessage "Starting Firewall"
    sh /etc/rc.firewall > /dev/null
  fi
```

```
}

StopService ()
{
  ConsoleMessage "Stopping Firewall"
  /sbin/ipfw -f -q flush
}

RestartService ()
{
  StopService
  StartService
}

RunService "$1"
```

Additionally, we require a StartupParameters.plist file to tell the system when to start our script.

```
{
  Description     = "Firewall";
  Provides        = ("Firewall");
  Requires        = ("NetworkExtensions","Resolver");
  OrderPreference = "Late";
  Messages =
  {
    start = "Starting firewall";
    stop  = "Stopping firewall";
  };
}
```

The real work is undertaken by the /etc/rc.firewall script where the actual calls to ipfw are made. For a moment, let's just look at the Firewall script. The service will only start if the environment variable FIREWALL is set to YES. The advantage is that if FIREWALL is undefined, it will default to NO. This allows us to try out new firewall rules by running the /etc/rc.firewall script by hand. But if we have to reboot our computer, our rules will not be automatically loaded until we add the line FIREWALL=-YES- to the /etc/hostconfig file. This is a useful safety net when we are developing our own rules. Once we run our own rules, the firewall tab under services will not be usable until we run sudo ipfw flush to remove our rules. Finally, we need to fill in /etc/rc.firewall. A copy of rules from earlier will do the job. With the changes made to the /etc/hostconfig file, our custom-made firewall rules will be loaded as part of the normal boot sequence. We now know nearly all we need know to play with our own rules.

How to Test Your Firewall

To test your rules, you need a few resources on hand. First, and most important, you should be working at your Macintosh. If you manage to create a set of rules that stops you from accessing the Internet, you will still be able to open up a terminal, take down the rules, and reestablish your connections. Failing that, you can at least get to the reboot switch. If you are tempted, as I have been, to ssh into your computer from work and make a couple safe changes, you'll discover that you've locked yourself out and will have to wait until you get home before you can fix anything.

Second, you will need access to a second computer that is outside your network. My Macintosh is plugged into a router and exposed as a DMZ host. I can plug my laptop into the same router and access the services on my Macintosh as any other computer would. This allows me to check that the services that I think I am exposing are in fact available.

This second computer also allows us to run nmap, our third resource. nmap is a port-scanning tool with a considerable pedigree. We are using it here to establish which ports are visible and accessible to the outside world. We know what services we intended to be available, but it can't do any harm to check.

```
$ sudo nmap -T3 -vv -sS -p 1-65535 -P0 example.com
```

```
Starting nmap 3.55 ( http://www.insecure.org/nmap/ ) at
 2004-12-11 16:12 GMT
Host example.com (x.x.x.x) appears to be up ... good.
Initiating SYN Stealth Scan against example.com (x.x.x.x) at
 16:12
Adding open port 80/tcp
Adding open port 22/tcp
The SYN Stealth Scan took 3127 seconds to scan 65535 ports.
Interesting ports on example.com (x.x.x.x):
(The 65532 ports scanned but not shown below are in state:
 filtered)
PORT    STATE  SERVICE
22/tcp  open   ssh
80/tcp  open   http
427/tcp closed svrloc

nmap run completed -- 1 IP address (1 host up) scanned in
 3128.366 seconds
```

The only port numbers that turned up were the ones we set up, and port 427 is closed to the outside world. Everything is looking good. Nmap is a powerful tool with many, varied options and is a useful program to master.

Why Would You Write Your Own Rules?

Given that the firewall that comes with the Macintosh does such a good job and is well integrated with the services that you can run, why would you want to write your own rules? If you have a lone Macintosh that connects to the Internet via an Ethernet cable, then you really don't need to write your own rules. However, if your Macintosh is part of a small network, you may well have services that you only want local machines to connect to. This is the situation I have where I allow remote control of my Macintosh via VNC from local machines but do not want anyone else to know that I even run the service, let alone have access to it.

Playing with a firewall can be quite scary, especially when you see what is attacking you for every moment that you are connected to the Internet. Your most prudent course of action at this point is to remove any custom rules that you have written and let the Sharing Control Panel take care of things. It was doing a perfectly good job before you knew what it was dealing with and will continue to do so going forward.

Peter Hickman is currently working as a programmer for Semantico, which specializes in online reference works and Access Control Systems. When not programming or reading about programming he can be found sleeping