Introduction

Up until now, we've looked at stateless and stateful firewalls. Remember, stateless firewalls only have the features of a given *packet* to use as criteria for whether that packet should be passed, blocked, or logged. With a stateful firewall, in addition to the fields in that packet, we also have access to the kernel's table of open connections to use in deciding the fate of this packet.

There's a problem, though. Picture an attacker that has launched attacks against almost every port on our web server box for the past half hour. The firewall has successfully repelled all of them, but now the attacker turns her attentions to port 80. All of the hostile overflow attempts are let through unhindered. Why? Because the firewall ruleset allows all traffic to the web server through, and our firewall can't remember the fact that this IP address has been pounding all the other ports on the system.

What if we could tell the firewall to remember the IP address of attackers and block them for a short period of time following their last attack? By remembering their past actions, we can block incoming web server connections that would otherwise have been allowed.

Iptables modules

The firewall code in the current Linux kernel (http://www.netfilter.org) is called Iptables or Netfilter (while there is a technical distinction, they're equivalent names for this discussion). The crucial feature of this firewall is its modular design. You have the ability to add new types of tests to perform on a packet and actions to take on it. These tests and action modules can be added to a running kernel.

For example, lets say you want to discard all packets that have any ipv4 options set. The stock kernel doesn't have a test for this. However, someone has written a firewall module that can perform this test. Once the module is compiled for and inserted into the current kernel, the following firewall command:

```
iptables -A INPUT -m ipv4options --any-opt -j DROP
```

will discard these packets. Conveniently, there's even a new *target* module that can strip off these headers if compiled and invoked like so:

```
iptables -t mangle -A PREROUTING -j IPV4OPTSSTRIP
```

Note that the kernel, as booted, had neither the ability to recognize (the first command above) nor strip (the second command) ipv4 options. These capabilities were provided by the ipv4options and IPV4OPTSSTRIP loadable kernel modules (lowercase modules are tests and uppercase modules are targets, by convention).

Categorizing attackers with the *recent* module

We're going to make use of the recent firewall module. We need to make use of this in two ways; first, to *identify* the IP address of malicious attackers, and second, to *punish* them in some fashion.

We're going to use the following tests to identify the malicious traffic.

```
iptables -A INPUT -p tcp -d my.mail.server --dport 25 --tcp-flags ACK \
 ACK -m string --string "rcpt to: decode" \
 -j LOG --log-prefix " SID664 "
```

```
iptables −A INPUT −p tcp −d my.mail.server −−dport 25 −−tcp−flags ACK \
 ACK −m string −−string "rcpt to: decode" \
 −m recent −−name MAILPROBER −−set

iptables −A INPUT −p tcp −d my.mail.server −−dport 25 −−tcp−flags ACK \
 ACK −m string −−string "rcpt to: decode" \
 −j REJECT −−reject−with tcp−reset

# "SMTP sendmail 5.6.4 exploit" nocase−ignored arachnids,121 classtype:attempted−admin sid:664
```

This rule is from the http://www.stearns.org/snort2iptables/ project which provides roughly equivalent iptables firewall rules from the Snort IDS ruleset.

The first two lines of each command identify the malicious traffic. In this case, we have traffic header for the smtp port on our mail server with the ACK flag set and the phrase "rcpt to: decode" in the packet. The first rule simply logs the traffic to syslog. The second rule is the one that records the source address of this packet in a kernel table called MAILPROBERS. The final rule is the only one that actually decides the fate of this packet; the packet is discarded and a tcp reset is sent back to the sender.

The first and third rules are concerned with what to do with *this* packet. The second rule is solely interested in remembering the attackers IP address for future punishments. OK, let's punish them:

```
iptables −A INPUT −m recent −−name MAILPROBER −j DROP
```

This one is placed near the top of the firewall, but after any "iptables −A INPUT −m state −−state ESTABLISHED,RELATED −j ACCEPT" rules. The rule sees if the source address of a new packet is in the MAILPROBER IP address list in the kernel. If it is, the packet is discarded. Here's the first time we've been able to use someone's past actions to block their future connections.

The above punishment is a little harsh, though. The fact that someone tried an old exploit 3 hours ago doesn't mean that IP address should be blocked from any communication with anyone in our network forever, which is the end result of the above DROP rule. If for no other reason, the attack might have come from a dialup IP address; the next person to acquire that address may simply want to retrieve a web page, and is now blocked because his predecessor was mildy hostile a while back.

Lets tone down the punishment a little:

```
iptables −A INPUT −m recent −−name MAILPROBER −−seconds 180 −j DROP
```

We'll ignore the attcker for 3 minutes, but after that we'll allow more packets in and see if they'll play nice again.

See the http://www.stearns.org/snort2iptables/ website and the psd port scan detector module for some more ideas of traffic to seed the attackers lists in recent. I'd strongly discourage using anything but established tcp session traffic, even though a good portion of the snort2iptables ruleset is udp, icmp or other protocols as non−tcp protocols are easier to spoof than tcp.

The Marquis De Sade, or punishments galore

So far we've put in a three minute block on this individual. Here are some other punishments to consider for more or less severe offenses. Simply replace the "−j DROP" above with one of the following. Note that one can log any packet by simply putting in a a log rule (see above for an example) ahead of the rule that decides what to do with the packet.

... −j REJECT

Like DROP, but it lets the attacker know her connections are being shot down. In many ways, this is more network−friendly as you're closing at least one end of the conversation somewhat cleanly. REJECT has a number of different methods of shutting down connections; see "iptables −j REJECT −h" for a list with brief descriptions.

... −p tcp −m iplimit −−iplimit−above 2 −j REJECT −−reject−with tcp−reset
... −p udp −m iplimit −−iplimit−above 2 −j REJECT −−reject−with proto−unreach
... −m iplimit −−iplimit−above 2 −j REJECT −−reject−with host−unreach

This triplet using the iplimit module might be appropriate for people that open up large numbers of connections to your machine; getright users come to mind. The first and second connections from an IP address come in just fine, but third and future connections get shot down in a protocol appropriate way.

... −m random −−average 25 −j DROP

Up until now, when we've decided to shun someone, it has been entire. Instead of dropping all their packets, how about randomly dropping 25% of them? The built in retransmissions in TCP will keep the conversation going (as may application level reliability mechanisms in UDP), but this has the effect of slowing down the connversation in the same way that a lossy line will. Good for bandwidth hogs. :−) The nth module will perform the same task as random.

... −m state −−state NEW −j DNAT −−to−destination 1.2.3.4:5555

OK, we've identified this person as an attacker. How about shunting her future connections into the waiting honeypot on IP address 1.2.3.4, port 5555?

... −j ULOG

Instead of logging a short description of the packet to syslog, lets log the entire packet to userspace. In addition to the ulog module, you'll also need to set up the userspace application to receive the packets.

... −j MIRROR

I include this in the interest of being complete, but this target serves almost no functional purpose. Shipping the attackers packets back to them with source and destination IP addresses and ports swapped chews up bandwidth, lets them know that you're annoyed, fills up the connection table on any intermediate firewalls twice as fast, and might be grounds for them to claim *you* are portscanning *them*. *sigh*

Pitfalls and considerations

The above sounds great − get the firewall to rememeber attackers and apply an instant block or other punishment. It's not without its problems, though. You need to pay attention to the following if you would like to make use of this approach.

Spoofed source addresses: tcp, established only

We're basing our choice of whether to punish someone on their source IP address; its analagous to blocking incoming calls based on their caller ID number. Unlike that caller ID on the phone which should always match the number of the person calling, IP addresses can be trivially spoofed. Lets say Eve, the Evil attacker, knew you were using this technique to block all conversations with people who send even a single packet to port 27374 on your web server. Eve creates a bogus packet with its source IP address replaced with the IP address of your name server. Your firewall remembers this and dutifully blocks all conversations with − gulp − your name server. Oops, no more dns service for you.

If you limit your "Is this IP address attacking me?" rules to inspecting established tcp sessions, the risk of spoofed source addresses drops because the attacker needs to put in a good deal more effort to spoof these. Even with this extra caution, you need to realize that spoofing may still be possible.

Zombie servers (and clients with DROP)

When using the "−j REJECT" target, the firewall in the middle will send a notification to the attacker that the communication is being dropped or that the host is unreachable (even if it really is reachable). The attacker can now shut down that connection and quickly reclaim the memory used. The server, however, is left in the dark; it gets no signal that the connection is now dead and waits for a timeout to occur before closing the connection and freeing up the memory used.

If large numbers of connections are shut down in this fashion, you may find that a lot of servers are left hanging around in memory. With a small number of servers, those inactive pieces of code can be moved out into swap, tying up a small amount of disk bandwidth, but with a large number, you may exhaust the available memory, which can lead to killing random processes.

At the very least, you'll want to allocate more disk space for swap usage during a large attack. It may also be appropriate to instruct your Operating System kernel and/or applications to reduce their timeouts so they close connections more quickly.

Small recent ip list

By default, the module will only remember 100 attacking IP addresses per list (MAILPROBER, above, for example). On a system being actively attacked from a number of locations, this won't be enough. When the module is first inserted, you should think about how many IP's to remember and provide that on the command line with something like:

```
modprobe ipt_recent ip_list_tot=1000
```

You'll use more memory with this approach, but I'm all for having the firewall do more work for me in exchange for a inexpensive ram.

Manual viewing and manipulation with /proc

The proc filesystem holds a readable, and even modifiable, list of IP addresses in each of your lists. This allows you to manually add or remove IP's, clear the table, or do additional checks or logs from a userspace program.

William is an Open−Source developer, enthusiast, and advocate from New Hampshire, USA. His day job at SANS pays him to work on network security and Linux projects.