

Firewall Logging & Monitoring

Chris Brenton and Others

In this document:

- What kinds of events do firewall admins want to monitor?
- Network connection logs
- Configuration details for specific firewalls
- Sample messages for firewall events
- artificial ignorance: how-to guide
- Firewall log processing References

What kinds of events do firewall admins want to monitor?

Significant events on firewalls fall into three broad categories: critical system issues (hardware failures and the like), significant authorized administrative events (ruleset changes, administrator account changes), and network connection logs. In particular, we're interested in capturing the following events:

- host operating system log messages -- for the purposes of this document, we'll capture this data at the minimum severity (maximum verbosity) required to record system reboots, which will record other time-critical OS issues, too
- changes to network interfaces -- need to test whether or not the default OS logging captures this information, or if the firewall software records it somewhere (any invocation of UNIX ifconfig or the equivalent?)
- changes to firewall policy
- adds/deletes/changes of administrative accounts
- system compromises
- network connection logs, which include dropped and rejected connections, time/protocol/IP addr/s/usernames for allowed connections, maybe amount of data transferred

The observant firewall administrator will notice that this list contains more than just network connection information. Most firewall logging tools focus on network connection records because protecting network connections is the most obvious task performed by the firewall, and because they're typically in a predictable format. At least, logging formats are relatively stable on any given platform, operating system and firewall application.

However, tracking administrative changes on your firewalls is a vital component of system and security administration. And records of administrative changes are often hard to track down. So we'll include them in the device-specific notes below.

Network connection logs

When most system administrators think about firewall logs, they think about network connection logs. Because firewalls are gateways between networks of varying trust levels, they provide an obvious place to record information about network traffic.

Once a firewall is installed and configured with the appropriate ruleset, dropped connection logs require some additional processing to derive a useful summary of activity on your network. Extracting information like numbers of port scans, or top ten sources of dropped packets usually requires writing a script, although more firewall developers are including those sorts of summary reports in their

Firewall Logging & Monitoring

Chris Brenton and Others

default installations. We'll describe built-in network connection reporting as appropriate, as well as after-market log processing tools, in the device-specific notes.

Notes on Specific Firewalls

IPtables

Recording iptables Administrative Activity by Chris Brenton with kibitzing from tbird & Bill Stearns

tbird editorializes: iptables limits its impact on local logging by completely failing to record information on policy changes. Since we've decided we care about administrative activity, we're going to use sudo to record the actions of all users who have permission to reconfigure the firewall.

And before you bring up that it's possible for someone to get out of the sudocage, or the ease with which a local root user can eliminate the log evidence of their activity, remember two points:

1. The goal here is not primarily to prevent a malicious local user from escaping without creating an audit trail, or to prevent a malicious local user from committing unauthorized activity, but to create a record of accountable actions on the firewall for regular reporting. Or for answering the desperately important question, usually from senior firewall admin to junior: "Exactly what were you doing, before you didn't do anything and it all stopped working?"
2. We can make it slightly harder for a local user to completely cover their own tracks by remembering to configure system logging on the firewall host to log remotely, as well as locally. And then making sure that firewall administrators don't have write privileges on the remote loghost.

Chris Brenton writes:

We'll require firewall administrators to use sudo to make changes to the iptables configuration. Add an entry:

```
User_Alias FIREWALLADMIN = tbird, wstearns, cbrenton
```

to emphasize the (perhaps theoretical) distinction between a firewall administrator (someone who deals only with the iptables application) and the all-powerful system administrator (who can exercise all superuser rights) (if such a distinction exists in my production environment). Then I'd do a:

```
FIREWALLADMIN ALL=(root) PASSWD: ALL: /sbin/iptables
```

Now at the command line I run:

```
[cbrenton@valhalla testing]$ sudo /sbin/iptables -A FORWARD -ieth0 -p tcp --tcp-flags ALL SYN,FIN -j LOG --log-prefix "SYNFINSCAN "
```

```
[cbrenton@valhalla testing]$ sudo /sbin/iptables -A FORWARD -ieth0 -p tcp --tcp-flags ALL SYN,FIN -j REJECT --reject-withicmp-host-unreachable
```

which I know worked because:

```
[cbrenton@valhalla cbrenton]$ sudo /sbin/iptables -L -n
```

```
Chain FORWARD (policy ACCEPT)
target prot opt source destination
LOG tcp -- 0.0.0.0/0 0.0.0.0/0 tcpflags:0x3F/0x03
LOG flags 0 level 4 prefix `SYNFINSCAN 'REJECT
```

Firewall Logging & Monitoring

Chris Brenton and Others

```
tcp -- 0.0.0.0/0 0.0.0.0/0
tcpflags:0x3F/0x03 reject-with icmp-host-unreachable
```

Running these commands generated the following log entries:

```
[root@valhalla root]# tail -3 /var/log/secure
```

```
Nov 11 07:06:36 valhalla sudo: cbrenton : TTY=pts/1 ; PWD=/home/cbrenton ; USER=root ;
COMMAND=/sbin/iptables -A FORWARD -i eth0 -p tcp--tcp-flags ALL SYN,FIN -j LOG --log-prefix
SYNFINSCAN
```

```
Nov 11 07:07:36 valhalla sudo: cbrenton : TTY=pts/1 ; PWD=/home/cbrenton ; USER=root ;
COMMAND=/sbin/iptables -A FORWARD -i eth0 -p tcp--tcp-flags ALL SYN,FIN -j REJECT --reject-with
icmp-hostunreachable
```

```
Nov 11 07:10:49 valhalla sudo: cbrenton : TTY=pts/1 ; PWD=/home/cbrenton/SANS/traces/testing ;
USER=root ;COMMAND=/sbin/iptables -L -n
```

So now `/var/log/secure` knows who ran the command and when, what switches were used, etc. We can let `syslog` dump this to a remote logging host.

Log Prefixing with iptables (by Chris Brenton)

Introduction

`iptables` is the built in firewalling tool available on any Linux system running kernel version 2.4 or later. While `iptables` is an extremely powerful firewall, and has many capabilities that are not even found in commercial firewalls, in this paper I'm going to focus specifically on `iptables`' ability to perform log prefixing. This paper assumes some prior knowledge of how `iptables` works. If you need to start with the basics, please review the `iptables` HOWTOs and F.A.Q.

If you're unfamiliar with the format of `iptables`' network connection logs, check out William Stearns' write-up.

This paper also assumes you have installed the latest stable Linux kernel (version 2.4.22 at the time of this writing) and the latest version of `iptables` (1.2.9) at the time of this writing). You can find the latest kernel at <http://www.kernel.org/> and the latest version of `iptables` at <http://www.netfilter.org>.

What is log prefixing?

Log prefixing is a feature that's so simple, people tend to overlook how powerful it can be. The prefixing capability allows you to define an `iptables` rule and specify a text string that should be recorded to the logs whenever that rule is matched. Let me give you an example. For the purposes of our discussion, assume `eth1` is the internal interface and `eth0` is the external interface:

```
iptables -A FORWARD -i eth1 -m state --state NEW -s192.168.1.0/24 -d 0/0 -j LOG --log-prefix "
OUTBOUND "
```

```
iptables -A FORWARD -i eth1 -m state --state NEW -s192.168.1.0/24 -d 0/0 -j ACCEPT
```

The first rule states

For traffic received on the `eth1` interface, match packets that are new connection requests originating from 192.168.1.0-192.168.1.255, going to any destination IP address. When a match is found, allow the connection request, log this packet and prefix the log entry with the text pattern " `OUTBOUND` ".

Here is an example log entry recorded by this pair of rules. Notice that the word `OUTBOUND` appears just before the packet specific information:

Firewall Logging & Monitoring

Chris Brenton and Others

```
Oct 31 06:11:35 gw1 kernel: ^ OUTBOUND IN=eth1 OUT=eth0SRC=192.168.1.101 DST=204.152.189.116
LEN=52 TOS=0x00 PREC=0x00 TTL=127 ID=18437 DF PROTO=TCP SPT=32865 DPT=80 WINDOW=5840
RES=0x00 SYN URGP=0
```

So with little fanfare we've just created a simple label that allows us to uniquely distinguish our outbound traffic from all other firewall log entries. Using `grep` we could then run the command:

```
grep OUTBOUND /var/log/messages > outbound-traffic.txt
```

The file `outbound-traffic.txt` would contain all of the log entries for traffic leaving our internal network and headed out to the Internet. If we are interested in seeing the traffic from one of our internal users, we can find the information in the file `outbound-traffic.txt`. Finding the information should be much easier than reviewing `/var/log/messages`, as the `outbound-traffic.txt` file will be much smaller and will only contain outbound traffic patterns.

Using log prefixing to simplify log review

The most time consuming task of any firewall administrator is reviewing the network connection logs. The first problem is the raw quantity of information that gets recorded. I've seen environments that collect 500 MB or more of log entries on a daily basis. The sheer number of log entries makes it difficult at best to do any type of a real forensic analysis.

The second problem is trying to identify off the top of your head what an attacker may be trying to do. Quick, quick, without looking it up, what is an attacker trying to access if you see a SYN packet going to 3128/TCP? What about a SYN to 2222/TCP? What about a connection attempt to 54321/UDP? Sure you can look up what these ports are used for, but that takes time (thus dragging out the log review process). Also, if you don't see the traffic patterns again for a month or two, you will probably forget what service is associated with these ports and you would have to look them up again.

Log prefixing can help cure both of these problems. We can use prefixing to label observed traffic patterns, which solves problem #2. We can then segregate traffic based on log prefixes to make dealing with problem #1 much easier.

Start with what you understand

A great place to begin with log prefixing is to label all of the traffic patterns you expect and understand. Here are some examples:

```
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth0 -d 1.2.3.4 --dport 22 -j LOG --log-prefix "
INBOUND_SSH "
iptables -A FORWARD -p tcp ^ --tcp-flags ALL SYN -i eth0 -d 1.2.3.5 --dport 25 -j LOG --log-prefix "
INBOUND_SSMTP "
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth0 -d 1.2.3.6--dport 80 -j LOG --log-prefix "
INBOUND_HTTP "
```

These three rules look for SYN packets going to ports 22, 25 and 80 at three different IP addresses. Let's assume these are all traffic patterns we wish to permit. Note that we would need to include an ACCEPT rule as we did in the first example, however I did not show them here as we are focusing on the prefixing capability.

What we've just done is uniquely identify each of these traffic patterns and distinguished them from all other log entries we may record. For example, while we could simply search our logs looking for 'DPT=22' to see all SSH traffic, this would show us all SSH traffic, both inbound and outbound. By using prefixing we have a simple way of distinguishing between these two directions. If we wanted to uniquely identify outbound SSH traffic, we could create the rule:

Firewall Logging & Monitoring

Chris Brenton and Others

```
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth1 -d 0/0 -  
dport 22 -j LOG --log-prefix " OUTBOUND_SSH "
```

Now, let's assume we only want to permit inbound SSH to the one IP address listed above. Consider the following logging rule:

```
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth0 -d 0/0 -  
dport 22 -j LOG --log-prefix " SSH_SCAN "
```

We place this rule after all the ACCEPT rules for inbound SSH. What we are doing here is uniquely identifying all inbound SSH connection attempts that are going to one or more IP addresses that have not been expressly permitted to receive this traffic pattern. The thinking is, if an IP address out on the Internet is trying to connect to 22/TCP on a system that is not permitted to receive that traffic pattern, the IP on the Internet is possibly hostile and probing for SSH servers. Chances are they have some exploit that works against some version of SSH, and they are looking for vulnerable servers.

So let's say we're doing a log review and wish to see if anyone is looking for vulnerable SSH servers.

We can run the command:

```
grep SSH_SCAN /var/log/messages > ssh-scans.txt
```

The file ssh-scans.txt will now contain all SSH probes against our environment, since legitimate inbound and outbound SSH traffic will be prefixed with a different label. In other words, we don't have everything lumped together like we did when we simply searched for traffic going to destination port 22. All SSH traffic is nicely segregated and easier to review.

So now we can extend this line of thinking to all other ports we permit through our perimeter. For example after we create our acceptable inbound SMTP and HTTP rules we could include:

```
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth0 -d 0/0 - dport 25 -j LOG --log-prefix "  
SMTP_SCAN "  
iptables -A FORWARD -p tcp --tcp-flags ALL SYN -i eth0 -d 0/0 - dport 80 -j LOG --log-prefix "  
HTTP_SCAN "
```

Now we can easily identify IPs on the Internet probing for these services as well.

Note that we don't have to actually offer the service to use this method. For example even if we don't permit POP-3 inbound across our perimeter, the rule:

```
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN -d 0/0 - dport 110 -j LOG --log-prefix "  
POP3SCAN "
```

would uniquely identify people probing for POP-3 servers.

Sneaky alerting with log prefixing

Let's assume we have a three legged firewall, with all Internet accessible services located off of the third network interface (eth2) on an isolated service network or DMZ. Clearly this network is one of our biggest points of vulnerability, as systems are directly accessible from the Internet. We can use log prefixing to help us identify if any of these systems become compromised.

When an attacker breaks into a server, one of the first things they do is transfer their toolkit. This usually includes a rootkit, as well as additional tools they use to cover their tracks. Usually the toolkit is transferred by generating an outbound FTP, TFTP or HTTP session to some other compromised host on the Internet.

Firewall Logging & Monitoring

Chris Brenton and Others

So here is what we do. We create ACCEPT rules for all patterns leaving the service network that we wish to permit (25/TCP from our mail server, 53/TCP and UDP/53 from our DNS servers, etc.). Then we create the following set of rules:

```
iptables -A FORWARD -i eth2 -j LOG " SERVICE_NET_COMPROMISE A "  
iptables -A FORWARD -j DROP
```

This rule identifies the philosophy that "any unauthorized traffic attempting to leave the service network could be an indication that a host has been compromised", since we've been good firewall administrators and expressly permitted only the traffic we expect to see. We can now set up some form of automated log scanning tool (such as swatch) to monitor /var/log/messages and notify us if the string SERVICE_NET_COMPROMISE is seen. Obviously we would want to respond to this situation as quickly as possible.

Identifying odd TCP flag combinations

It's not uncommon to see hostile IPs on the Internet sending you traffic with odd TCP flag combinations. An odd TCP flag combination is considered to be any flag combination that is not explicitly defined in the RFCs, and could potentially be used for malicious intent. Odd TCP flags frequently reveal someone performing active fingerprinting, or possibly trying to circumvent your firewall. We can easily use log prefixing to uniquely identify these traffic patterns. Consider the following rules:

```
iptables -A FORWARD -p tcp --tcp-flags ALL SYN,FIN -j LOG --log prefix " SYNFINSCAN "  
iptables -A FORWARD -p tcp --tcp-flags ALL FIN -j LOG --log prefix " FINSCAN "  
iptables -A FORWARD -p tcp --tcp-flags ALL NONE -j LOG --log prefix " NULLSCAN "  
iptables -A FORWARD -p tcp --tcp-flags ALL FIN,PSH,URG -j LOG - log-prefix " XMAS "
```

The first rule looks for traffic with the SYN and FIN flags turned on. This flag combination is frequently used by people trying to sneak their way past a firewall or router that bases its accept-or-deny decision on rudimentary packet filtering. The second rule looks for packets with only the FIN flag set. This is usually an indication that someone is trying to perform a stealthy port scan of a UNIX system. The last two rules look for flag combinations associated with someone trying to actively fingerprint your system.

By uniquely labeling each of these flag patterns, we can better organize the firewall output for log review. We should also follow each of these rules with an appropriate DROP or REJECT rule, to keep this traffic from passing our perimeter. Consider this additional example:

```
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN,ACK -m state --state NEW -j LOG --log-prefix "  
SPOOF_FALLOUT "  
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN,ACK -m state --state NEW -j REJECT  
--reject-with icmp-host-unreachable
```

The first rule looks for TCP packets with the SYN and ACK flag set that are not part of a previous outbound session. Unsolicited SYN/ACK responses are one sort of backscatter signature, because they're triggered when an evildoer forges our legal IP address as the source of a SYN flooding attack against some third party. The system under attack sends responses to the attack packets, which are routed in good Internet fashion to our machine (since our address was forged as the source of the malicious traffic). So this prefixing rule will allow us to see how often our address space is getting spoofed as part of these attacks.

Now, note the second rule, which tells iptables how to handle responses to spoofed packets. Rather than just dropping the traffic, we are rejecting it with an ICMP host unreachable. This helps minimize the effect of the attack against the victim machine, because it helps clear the victim's connection

Firewall Logging & Monitoring

Chris Brenton and Others

queue of the bogus packet. If we just drop this packet, the remote host's connection queue stays tied up for up to two minutes. Rejecting the packet makes us a good Internet neighbor, and reduces the effectiveness of spoofing our address space as part of an attack.

Identifying commonly probed ports

Once we have uniquely labeled all of the well known services we want to keep track of, we can then turn our attention to labeling some of the Trojan and back door ports we see getting probed as well. For example:

```
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN -s 0/0 - sport 31789 -d 0/0 --dport 31789 -j LOG --log-prefix " HACKATAACK "  
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN -d 0/0 - dport 12345:12346 -j LOG --log-prefix " NETBUS "  
iptables -A FORWARD -i eth0 -p tcp --tcp-flags ALL SYN -d 0/0 - dport 17300 -j LOG --log-prefix " KUANG2_SCAN "  
iptables -A FORWARD -i eth0 -p udp -s 0/0 --sport 1024:65535 -d 0/0 --dport 54321 -j LOG --log-prefix " BO2K "
```

Note that for TCP-based back doors, we match on the SYN flag being the only flag set, as well as the port number typically used by the program. Properly matching UDP-based back doors is a bit more difficult because we don't have state flags to review. In this example, the last rule looks for attempts to access Back Orifice 2000. We search for a high-numbered source port, in conjunction with a destination port of TCP/54321, the well known port used by BO2K. Requiring these source and destination ports, as well as processing this UDP rule after the TCP state table, should help eliminate any BO2k false positives.

Using prefixing to spot patterns

Prefixing your log entries and organizing them by labels hugely enhances your ability to identify trends that you may not be able to spot if you look at one huge file full of firewall network connection logs. For example, consider the following firewall log entries:

```
Nov 2 07:09:23 gw1 kernel: KUANG2_SCAN IN=eth0 OUT= eth1 SRC=x.x.  
x.182 DST=1.2.3.4 LEN=60 TOS=0x10 PREC=0x00 TTL=52 ID=63731 DF PROTO=TCP SPT=48198  
DPT=17300 WINDOW=5840 RES=0x00 CWR ECE SYN URGP=0 Nov 2 07:20:17 gw1 kernel:  
KUANG2_SCAN IN=eth0 OUT=eth1 SRC=x.x.  
x.182 DST=1.2.3.5 LEN=60 TOS=0x10 PREC=0x00 TTL=51 ID=63037 DF PROTO=TCP SPT=48215  
DPT=17300 WINDOW=5840 RES=0x00 CWR ECE SYN URGP=0 Nov 2 07:29:55 gw1 kernel:  
KUANG2_SCAN IN=eth0 OUT=eth1 SRC=x.x.  
x.182 DST=1.2.3.6 LEN=60 TOS=0x10 PREC=0x00 TTL=51 ID=47856 DF PROTO=TCP SPT=48234  
DPT=17300 WINDOW=5840 RES=0x00 CWR ECE SYN URGP=0 Nov 2 07:39:53 gw1 kernel:  
KUANG2_SCAN IN=eth0 OUT=eth1 SRC=x.x.  
x.182 DST=1.2.3.7 LEN=60 TOS=0x10 PREC=0x00 TTL=51 ID=55002 DF PROTO=TCP SPT=48250  
DPT=17300 WINDOW=5840 RES=0x00 CWR ECE SYN URGP=0
```

We have a single IP address out on the Internet that appears to be probing for the KUANG2 back door on each of our internal systems. Now, note the time interval. These probe packets are hitting our perimeter about once every 10 minutes. This is a very slow scan. So slow, in fact, that you would be likely to miss it during a review of the raw log (depending of course on how you do your review), because there would be many other log entries interspersed between these entries. By running the following command:

```
grep KUANG2_SCAN /var/log/messages > kuang2.txt
```

we can remove all those other log entries and this traffic pattern sticks out like a sore thumb.

Firewall Logging & Monitoring

Chris Brenton and Others

[third editorializes: We discuss a variety of other tricks for pulling interesting patterns out of firewall logs later on in this document.]

How to process your logs

It should be clear that labelling log entries is only half the work: you still gotta read the pesky things. You should also consider automating the process of segregating each of these labels into their own files. I have written a paper entitled "Reviewing your logs with grep" that should help you set up this process.

Recording the process that generated an outbound connection request (by Bill Stearns; taken from posting to Firewall-Wizards mailing list, 15 November 2003)

On Fri, 14 Nov 2003, Chris de Vidal wrote:

I have several rules like this:

```
/sbin/iptables --append OUTPUT --jump LOG --log-level DEBUG --logprefix "OUTPUT packet died: "
```

at the bottom of my OUTPUT chain to debug which outgoing packets get dropped so I can adjust the rules as necessary. It's been working well for months.

Trouble is I don't always know which program is producing these packets. It would be handy to also see the PID and/or program name responsible for these packets. Any idea how?

The "owner" match module could be used to check what application/UID created the packet. This can only be used in the OUTPUT and POSTROUTING chains, but that's perfect for what you need. To use it, get a list of all applications - clients or servers - that might be running at a given time. Then put in these rules instead of the one you listed above:

```
for App in sshd gabber httpd netscape-communicator named ;
do/sbin/iptables --append OUTPUT -m owner --cmd-owner "$App" --jump LOG --log-level DEBUG
--log-prefix "OUTPUT $App packet died: "done /sbin/iptables --append OUTPUT -m owner --cmd-owner
$App --jump LOG --log-level DEBUG --log-prefix "OUTPUT packet died: "
```

To get a quick list of candidate applications, try:

```
ls -al /proc/[0-9]*/exe 2>/dev/null | sed -e 's@.*!/@@' | sort | uniq | grep -v 'exe'
```

For reference, here's the syntax for the module:

```
OWNER match v1.2.8-20030601 options:[!] --uid-owner userid Match local uid[!] --gid-owner groupid
Match local gid[!] --pid-owner processid Match local pid[!] --sid-owner sessionid Match local sid[!]
--cmd-owner name Match local command name
```

Bill Stearns' document on logging entire packets to syslog using iptables

FireWall-1

We'll use the FireWall-1 host operating system syslog data for application errors, hardware failures, and other pertinent information. We'll use FW-1's auditing to monitor changes to the firewall policy, whether made through the GUI administration tool or the OS command line. And we'll document the configuration requirements to get FW-1 network connection logs to syslog.

FireWall-1 implements a distributed security architecture, optimized for centralized management of a large number of firewall devices and other network equipment (such as routers and VPN servers). The core components of FW-1 are the FireWall Module and the Management Server. The Management Server provides point-and-click access to the security policies for the FireWall Modules

Firewall Logging & Monitoring

Chris Brenton and Others

it manages. The FireWall Modules apply specific policies to the appropriate network traffic. In small FireWall-1 installations, both the FireWall Module and the Management Server are run on the same host operating system and platform, so syslog and network connection log forwarding are configured on the same physical device. In remote management FireWall-1 installs, the network connection logs must be sent to the loghost from the Management Server; syslog and security policy changes are forwarded from the individual FireWall modules.

If you're one of the lucky few people who manage their FW-1 systems through the command line, records of policy saves and loads are recorded to the local system's syslog via the FW-1 application (well done, Checkpoint).

If you manage your FW-1 infrastructure using the GUI application, as the vast majority of FW-1 administrators do, you'll have to do a bit more work to get records of administrative activity into your logging infrastructure. Verify that \$FWDIR is properly defined, and determine the location of the UNIX logger utility. The file \$FWDIR/log/cpmmgmt.aud contains records of policy changes and administrative access to the firewall via the GUI. To get that data to syslog, issue the following commands as root:

```
/bin/sh
```

```
/bin/tail -f $FWDIR/log/cpmmgmt.aud | /bin/logger -p local6.info > /dev/null 2>&1 &
```

[Note that these commands must be run from /bin/sh for the redirect to work.] You're of course welcome to use your own favorite shell, assign the appropriate facility and level to the firewall data, customize to your heart's content. But remember, once you've got the data flowing into syslog, to add the appropriate script to your start-up configuration so that it starts running once FW-1 itself is alive.

Getting FW-1 network connection logs to syslog (posted to the Dragon mailing list by Scott Wozny of Enterasys, and brought to my attention by former student Jeffrey Stebelton):

- Go into Security Policy Manager and go to the Policy -> Properties -> Log and Alert.
- Change the User Defined Command to /bin/logger -p daemon.notice (replace /bin/logger with the appropriate path for your firewall if necessary)
- Select the rules in your firewall policy that you'd like sent to syslog, and change their Track column to User Defined.
- Save and apply the rulebase.

To verify that you've started forwarding your network connection logs to syslog, check your central loghost (or the firewall's local log repository) for network connection entries. Here's what a FW-1 connection log looks like, pumped into syslog:

```
Feb 26 17:47:27 moose.sj.counterpane.com root: 17:47:27 dropmoose >hme0 proto tcp src elendil dst moose service http s_port58209 len 44 rule 4 Feb 26 17:47:28 moose.sj.counterpane.com root: 17:47:27 dropmoose >hme0 proto udp src elendil dst moose service domain s_port58231 len 90 rule 4 Feb 26 17:47:28 moose.sj.counterpane.com root: 17:47:27 dropmoose >hme0 proto udp src elendil dst moose service domain s_port4001 len 82 rule 4 Feb 26 17:47:28 moose.sj.counterpane.com root: 17:47:27 dropmoose >hme0 proto tcp src elendil dst moose service X11 s_port58232 len 44 rule 4
```

Cisco PIX

Netscreen

Sample messages for firewall events
artificial ignorance: how-to guide

tbird editorializes: The right way to deal with your logs is to parse the data, such that events you know are not time critical are filtered out (I'll call this the ignored category) events that are time critical are used to create alerts as quickly as possible (the alarmed category) events that don't fit into either category are read by a human on a regular basis and then designated as ignored or alarmed

Firewall Logging & Monitoring

Chris Brenton and Others

Here's the original proposition, posted by Marcus Ranum to the Firewall-Wizards mailing list on Tuesday, 23 Sep 1997 23:06:06 [I've modified the formatting a bit to make it a little easier to read, but the content is all Marcus]:

By request, here's a quick how-to on log scanning via artificial ignorance. :) It assumes UNIX and the presence of a good grep- you could use other stuff if you wanted to but this is just an example. Setting up a filter is a process of constant tuning. First you build a file of common strings that aren't interesting, and, as new uninteresting things happen, you add them to the file. I start with a shell command like this:

```
cd /var/logcat * | sed -e 's/^.*demo//' -e 's/[[0-9]*]// | sort | uniq -c | sort -r -n > /tmp/xx
```

In this example "demo" is my laptop's name, and I use it in the sedcommand to strip out the leading lines of syslog messages so that I lose the date/timestamps. This means that the overall variation in the text is reduced considerably. The next argument to sedstrips out the PID from the daemon, another source of text variation. We then sort it, collapse duplicates into a count, then sort the count numerically.

This yields a file of the frequency with which something shows up in syslog (more or less):

```
297 cron: (root) CMD (/usr/bin/at)167 sendmail: alias database /etc/aliases.db out of date120 ftpd:
PORT61 lpd: restarted48 kernel: wdpi0: transfer size=2048 intr cmd DRQ ... etc
```

In the example on "demo" this reduced 3982 lines of syslog records to 889.

Then what you want to do is trim from BOTH ends of the file and build an "ignore this" list. In this example, I don't care that cronran "at" OK so I'd add a regexp like:

```
cron.*: (root) CMD (/usr/bin/at)
```

That's a pretty precise one. :)

At the bottom of my file there were about 200 entries that looked like:

```
1 ftpd: RETR pic9.jpg1 ftpd: RETR pic8.jpg1 ftpd: RETR pic7.jpg1 ftpd: RETR pic6.jpg
```

Clearly these are highly unique events but also not interesting. So I add patterns that look like:

```
ftpd.*: RETRftpd.*: STORftpd.*: CWDftpd.*: USERftpd.*: FTP LOGIN FROM
```

Now, you apply your stop-list as follows:

```
cat * | grep -v -f stoplist | sort, etc --
```

This time I get 744 lines. Putting a pattern in that matches:

```
sendmail.*: .to=
```

drops it down to 120 lines. Just keep doing this and pretty soon you'll have a set of patterns that make your whole syslog output disappear. You'll notice that in the early example I had a warning from sendmail because the aliasesdatabase was out of date. Rather than putting a pattern for that, I simply ran newalias. Next time my aliasesdatabase is out of date, my log scanner will tell me.

System reboots are cool, too. My log shows:

Firewall Logging & Monitoring

Chris Brenton and Others

```
48 kernel: wdc2 at pcmcia0: PCCARD IDE disk controller
48 kernel: wdc1 at pcmcia0: PCCARD IDE disk controller
48 kernel: wdc0 at isa0 iobase 0x1f0 irq 14: disk controller
48 kernel: wd0 at wdc0 drive 0: sec/int=4 2818368*512
```

...

Those will be pretty much static. So I add those exact lines. Now they won't show up whenever the system boots. BUT I'll get a notification if a new SCSI drive is added, or (I did this deliberately!):

```
kernel: fd0c: hard error writing fsbn 1 of 1-19 (fd0 bn 1; cn
kernel: fd0: write protected
```

Oooh! Some bad boy trying to step on my tripwire file!

Or:

```
kernel: changing root device to wd1a
```

..interesting. My pattern was for wd0a!

I used to run this kind of stuff on a firewall that I used to manage. One day its hard disk burned up and my log scan cheerfully found these new messages about bad block replacement and sent them to me. :) The advantage of this approach is that it's dumb, it's cheap -- and it catches stuff you don't know about already.

Once you've got your pattern file tuned, put it in cron or whatever, so it runs often. The TIS Gauntlet has a hack I wrote called "retail" which I can't unfortunately release the code for, but is easy to implement. Basically, it was like tailbut it remembered the offset in the file from the previous run, and the inode of the file (so it'd detect file shifts) - the trick is to keep one fd open to the file and seek within it, then stat it every so often to see if the file has grown or changed inode. If it has, read to EOF, open the new file, and start again. That way you can chop the end of the log file through a filter every couple seconds with minimal expense in CPU and disk I/O.

I'm sure there are lots of fun ways this simple trick can be enhanced -- but just in its naive form I've found it quite useful. I wish I had a program that helped me statistically build my noise filters, but in general I find it's about a 2 hour job, tops, and it's one you do once and forget about.

tbird editorializes again: It's a two hour job on a very small network or for a very small number of monitored machines. I asked mjr about this once, and he pointed out that networks were a lot less complicated in 1997 than they are now.

Firewall log processing

Reviewing Firewall logs with grep(by Chris Brenton)

Introduction

One of the most difficult and time consuming parts of maintaining a secure network perimeter is reviewing firewall logs. It's not uncommon for an organization to generate 10, 50, 500 MB or more worth of firewall log entries on a daily basis. The task is so daunting, in fact, that many administrators stick their heads in the sand and simply do not review their logs. No matter how tempting, this is a mistake: when an attacker compromises your perimeter security, you don't want to turn a blind eye.

In this paper I'll discuss setting up an automated process in order to help sort your firewall log files and expedite a daily log review. The automated process will break down your firewall log into "bins" (a bin is a unique file of similar information) that are broken out by what ever method of categorization you wish to use. One of the strengths of this process is that you can customize it to make log review easier for you and your environment. For example, some people find it easier to organize their bins

Firewall Logging & Monitoring

Chris Brenton and Others

based on IP address, while others want their bins sorted by service. Some people may want odd flag combination segregated into their own unique bins, while others don't want that granular an analysis. So concentrate on the actual process being discussed, not how and in what order I chose to break up the data in this paper.

Also in this paper we will be working heavily with `grep`, a UNIX utility that allows you to specify a pattern and then search one or more files to see if they contain that pattern. Many people reading this paper will be thinking "Hey, this would be much easier using Perl/awk/sed/etc./etc./etc.". I chose to use `grep` for a number of reasons. First, it's very simple to use. While Perl may be more efficient for processing log files, using Perl also has a much steeper learning curve. I can teach the basics of `grep` in a few paragraphs, but teaching Perl is well outside the scope of a paper like this. Also, `grep` is a little easier to acquire and maintain. There are no security issues or environment tweaks. `grep` ships with all modern UNIX environments ; a freeware version for Windows can be downloaded from <http://www.tgries.de/agrep/> Simply make sure `grep` is in your path and you are ready to go.

If you prefer to use Perl or any other pattern matching tool, please feel free!

Also, while this method will work with the network connection logs generated by any firewall that stores its logs in a text format, the process is much easier if you have a firewall that allows you to record additional text information to the log file entries. A good example is `iptables`' log prefixing capability. I talk about the strengths of log prefixing in another paper titled "Log Prefixing With `iptables`".

Working with `grep`

As mentioned, `grep` allows you to search one or more files looking for a specific pattern. I highly recommend you spend some time learning `grep`, as it has multiple uses beyond the ability to help organize your firewall log files. I'm only going to cover the basics here that you will need to get started. `grep`'s syntax is pretty straight forward. The command:

```
grep foo firewall.log
```

will search for all instances of "foo" within the file `firewall.log`. Any time a match is found, the matching line is printed to the screen. Note that the string "FOO" would not be matched as the search is case sensitive. To match all case variations of the string "foo", we would use the `-i` switch like this:

```
grep -i foo firewall.log
```

If you would get the results from this search into a file than have them listed on the screen, simply use a greater than sign followed by the path and file name:

```
grep -i foo firewall.log > c:\logs\foo.txt
```

If you want to match all lines that do not contain the character string "foo", use the `-v` switch:

```
grep -v foo firewall.log
```

Note that we can combine this with the `-i` switch to make the pattern match case insensitive.

You can also use wild cards to match on multiple patterns. `grep` uses a period as a wild card character.

So the command:

```
grep .53 firewall.log
```

Firewall Logging & Monitoring

Chris Brenton and Others

matches lines that contain .53, but also 053, 153, 253, A53, a53, etc. as well. This can cause problems when processing firewall logs, as you might want to match on an actual period character (say as part of an IP address). To tell grepto search for a period, rather than interpreting the period as a wild card, simply precede it with a backslash:

```
grep \.53 firewall.log
```

The above command would only match on ".53".

If part of the string you are specifying contains space characters, encase the search string in quotes. So for example:

```
grep 'match this string' firewall.log
```

Note that on Windows double quotes are used. So to run this command on Windows you would type:
grep "match this string" firewall.log

Finally, let's say you want to match on multiple patterns on the same line, but the patterns are not necessarily all lined up. For example maybe you want to see DNS zone transfers but you are not interested in seeing DNS queries. Both communicate using a target port of 53, but zone transfers always use 53/TCP while queries usually use 53/UDP. Let's further assume that the firewall records the transport (TCP) at the beginning of the log entry but the port number (53) at the end of the line. To perform multiple matches we simply use multiple grepstatements and pipe them together. For example:

```
grep \.53 firewall.log | grep -i tcp
```

should match all 53/TCP traffic, depending on our logging format.

Again, this is just an introduction to grepthat will get you through the rest of the information in this paper. I highly recommend that you spend some time experimenting with this tool.

Firewall log formats

Exactly how to format your grepcommands is going to vary based on the output of your firewall. There is no "standard" method of formatting log entries; every firewall does it a bit differently. For example, here's an example of a Firewall-1 log entry:

```
15:35:11,N1001,drop,http,1.2.3.4,2.3.4.5,tcp,8,3748,len 29
```

The native FW-1 connection logs are stored as binary files in a binary format.. This particular entry has been exported to ASCII. Note that everything is comma separated and there is very little information to work with. On the plus side, services are normally identified by their well known name, which makes pattern matching and interpretation a bit easier.

This is an example of a Cisco PIX connection message:

```
Nov 13 21:02:28 66.33.141.81 %PIX-2-106001: Inbound TCP connection denied from 1.2.3.4/56086 to 2.3.4.5/17300 flags SYN on interface outside
```

Note that the IPs and port number are in close proximity to each other, which makes matching on a combination of these values a bit easier.

Here is an example of a Netscreen firewall log entry:

```
2002-07-23 16:19:09 Permit 192.168.1.2:33061->192.168.1.2:33061>2.3.4.1:2044 59 sec TCP PORT 2044
```

Firewall Logging & Monitoring

Chris Brenton and Others

Again we have IPs and port numbers identified in close proximity to each other. We also have a nice text string to match on when looking for a specific target port.

Finally, here is an iptables log entry, created without using log prefixing:

```
Nov 2 19:54:47 gw1 kernel: IN=eth0 OUT=eth1 SRC=1.2.3.4
DST=2.3.4.5 LEN=62 TOS=0x00 PREC=0x00 TTL=43 ID=20707 DF
PROTO=UDP SPT=56455 DPT=53 LEN=42
```

Note that each field value is preceded by a label description. This is by far the easiest type of logging format to pattern match against.

So when I show `grep` examples in this paper, I will try to use a generic command, "nearly" functional with all these logging formats. You will probably have to do some tweaking, however, to get it to work with your specific firewall logs.

Organizing a firewall log

Now that we have the basics out of the way, let's turn to the actual process. Rather than reviewing all of our firewall log entries line by line based on the time they were recorded, we are going to use `grep` to isolate log entries based on this secondary sort criteria.

You should start by screening out authorized traffic patterns. Since these are a known entity, let's get them out of the way first. This may include things like 80/TCP traffic going to your Web server, or 25/TCP traffic to and from your SMTP server.

We will typically process traffic in two passes, first creating a file that contains all data that matches the pattern of interest, and then creating a separate file that contains all data not matching this pattern -- both of these may be used for further processing depending on what you're trying to accomplish. For instance, if the IP address 1.2.3.4 is being used by our Web server.

```
grep '1.2.3.4/80' firewall.log > web-server1.txt
grep -v '1.2.3.4/80' firewall.log > temp-file1.txt
```

The first line grabs all traffic going to and from port 80 on our Web server and records it in the file `webserver1.txt`. In the second pass we match on all the other log entries and write them to a temporary working file called `temp-file1.txt`. The reason for creating the working file is that we will run our next set of `grep` commands against this file instead of the original `firewall.log` file. Since `temp-file1.txt` is smaller than the original, this will speed up our file sorting process.

So let's say the next thing we're tracking is SMTP traffic to our mail server. To bin this traffic we would use a similar format:

```
grep '1.2.3.5/25' temp-file1.txt > inbound-smtp.txt
grep -v '1.2.3.5/25' temp-file1.txt > temp-file2.txt
```

Again, our first line records all inbound SMTP traffic and puts it in a file by itself. Note that we did not search the original `firewall.log` file, but rather the working file we created in the first set of commands. In the second command we again grab everything except the inbound SMTP traffic, and create a new (even smaller) temporary working file.

Next, let's say we want to create another file that records all outbound SMTP traffic. This gets a little bit sticky because we need to match random upper ports coming from our SMTP server's IP address, going to port 25 on random IPs on the Internet. In other words, we don't have a long pattern string to match on, but rather two shorter ones. The following command should do the trick:

Firewall Logging & Monitoring

Chris Brenton and Others

```
grep '1.2.3.4' temp-file2.txt | grep '/25 ' > outbound-smtp.txt
```

I say this "should" do the trick because we are matching on the SMTP servers IP address as well the string "/25 ". If however our logs contain another instance of the pattern "/25 ", say as part of a date field, we will incorrectly match those lines as well and we'll have data getting stored to the wrong bin. Note that this format would work fine with the PIX and Netscreen logs shown above, but your firewall format may vary.

Now, normally we have a second line that grabs all other traffic and creates a new working file. In this situation we have a problem however because we used a combination of two grep commands to capture the data. In other words, simply performing a:

```
grep -v '1.2.3.4' temp-file2.txt | grep -v '/25 ' > temp-file3.txt
```

WILL NOT WORK. We've previously grabbed inbound and outbound port 25 traffic to this server. What if someone did a port scan against our mail server? The first grep command grabs all traffic that does not involve the IP address 1.2.3.4, which means we would end up ignoring all the port scan log entries against this host. We would also miss SSH or any other non-SMTP sessions. Clearly we have to bin this traffic before we remove all log entries involving our SMTP server.

```
grep '1.2.3.4' temp-file2.txt | grep -v '/25 ' > non-smtp mailserver.txt
```

Now that we have categorized all the traffic to our mail server, we can remove all entries and create a new working file:

```
grep -v '1.2.3.4' temp-file2.txt > temp-file3.txt
```

Note that clean up here was a bit of a pain. This is because we used two grep commands strung together. While this functionality can be useful for categorizing traffic, it can be a pain to clean up after.

Just take your time and think about the impact of all of your commands. One way you can go about checking your work is to perform a diff or a file compare between the working files. The only differences should be the data that you write to the bin files. If any additional data has gone missing, you have a problem with your script. Note that you only need to worry about this during the initial debugging process. Once your script is stable you can ignore this check.

So far we've assumed that you want to break things out by individual service going to each IP address. What if you don't want that level of granularity? For example what if we only wish to see the initial SYN packet on all inbound SMTP connections lumped together in the same file, regardless of which internal IP the traffic is headed to? Certainly we could do that instead. For example with a PIX, you could use a command similar to the following:

```
grep '/25 flags SYN' temp-file1.txt | grep 'Inbound TCP' > inbound-smtp.txt
```

In fact, this format can come in handy when looking for scanners. Let's assume that we do not have any FTP servers that are accessible from the Internet. That, or we have already placed all legitimate FTP traffic into its appropriate bin. Consider the following command:

```
grep '/21 flags SYN' temp-file5.txt | grep 'Inbound TCP' > ftp scan.txt
```

The concept is, if someone on the Internet is attempting to access the FTP port on one or more systems that do not legitimately offer the service, the user probably has evil intent. They may be

Firewall Logging & Monitoring

Chris Brenton and Others

looking for FTP servers they can exploit, or possibly looking for a spot to save their wares (stolen software). Either way, they are up to no good.

Now, personally I do not review my ftp-scan.txt file every day. I know that in my environment its "normal" for me to see an ftp-scan.txt file that is 150KB or less. This is because there is always some number of people on the Internet trolling for FTP servers. What motivates me to actually review the file is if I notice the file size has increased dramatically. If the file size increase is due to a large number of source IP addresses suddenly scanning for FTP servers, this could possibly be due to a new FTP exploit making the rounds, so I will investigate further in order to ensure that the FTP software I'm running is not vulnerable and the server itself has not been compromised.

This method can also be extended to look for other things, like people scanning for known back doors. For example the following grepcommand would check an iptables firewall log to see if people have been scanning for Kuang2:

```
grep 'DPT=17300' temp-file8.txt | grep ' SYN ' > kuang2-scan.txt
```

It should be clear that there is no "one right way" to go about organizing your data. You should use whatever format is going to make your life easier.

What to parse out and when

I had to think long and hard about including this section, because I'm a firm believer that you should parse the data in what ever order makes the most sense for you. I'm including this section as guidance for the people who are just getting started. This is what works for me, your mileage may vary. Please feel free to modify this order as you see fit.

TCP resets and ICMP unreachable

The first thing I always parse out is TCP resets and ICMP unreachable packets (type 3's and 11's). I find these interesting because they indicate a communication problem or someone performing a successful port scan. I break them out by direction, inbound vs. outbound, and these are always the first files I review.

Inbound known services

Next I parse out all inbound services that are offered by my environment. Services like HTTP, SMTP and DNS are perfect examples. I monitor the file sizes and usually do not bother to review these files line by line unless the file size is odd or a potential problem is indicated from one of the other files I review. I personally like to break out individual files for every service going to each unique IP address.

Outbound known services

Similar to inbound, I then parse out all expected outbound services. For service network systems (like my mail server and DNS) I break this out on a per service, per IP address fashion similar to how I handle inbound traffic. For all other IP addresses (like all my internal workstations) I break out HTTP, HTTPS, FTP, DNS, and "other". The first four tend to account for about 95% of all outbound traffic. I usually do not look at them unless I'm troubleshooting, find something odd, or the file size does not look right. The "other" file always gets reviewed just to see what types of odd traffic patterns are attempting to leave my network. Some environments may wish to actually review the HTTP file, and possibly even break it out based on source IP address. This depends on how granular you wish to get.

Known back doors

Next I parse out all of the known back doors, such as Kuang2 and BO2K. I'm not particularly interested in this traffic, so this removes known traffic patterns from the working file. Why look up these service ports over and over again when you can just categorize them and be done with it?

Firewall Logging & Monitoring

Chris Brenton and Others

Create an "interesting" file

When you have parsed out all of the traffic patterns you understand, you will probably be left with a working file that still contains some data. This could be a port scan against one or more of your systems, or possibly probes against a port that you have never seen before. This is "the good stuff", or the real meat that you are actually trying to identify within your logs. What I like to do is rename this last working file "interesting.txt" -- it's always one of the first files I review. If you find a pattern that you recognize and you wish to start categorizing it, simply add the appropriate grepcommands to your script. The idea is to make this file as small as possible -- you'll end up reviewing this one with a fine toothed comb.

Automating the process

Having covered how to parse the data, let's look at simplifying the process. You will want to create a batch file or shell script that includes all of your grepcommands for sifting through your firewall log. Once you have a script you are happy with, simply automate the process. You can set up a cronjob or a scheduled task that will launch your script and parse the data early every morning. That way when you arrive at the office, you can quickly review your bins to see if there is anything you need to worry about.

For example, here is a copy of my cronentry on my logging server:

```
25 5 * * * /usr/sbin/chkfwlogs
```

The shell script chkfwlogs runs every morning at 5:25 and parses the firewall log from the day before. By the time I arrive at work my bin files are ready to be reviewed for anything that warrants further action. I used to spend as much as four hours per day reviewing my firewall logs. Using this method, I've cut that time down to less than 20 minutes and I'm catching more of the interesting traffic that is hitting or passing my perimeter than I used to. Developing your script takes some time up front, but after 3-5 days however you should notice that your log review time has dropped dramatically.

A script example

I teach perimeter security for SANS. An important part of that class is learning to review log files. The following script is based on a class example in which students review the text output from a tcpdump file. I've added notes to describe what each line does. Feel free to borrow from this script and adopt it for your own environment.

```
#####  
# LOG ANALYSIS GREP SCRIPT  
#####
```

```
# Grab yesterdays log file, give it a useful name, and change to the working directory.
```

```
cp /var/log/messages.1 /root/logcheck/full_firewall.log  
cd /root/logcheck
```

```
# Resets are an indication of port scans or communication errors  
# which make them "interesting". Dump them all in the same file  
# so that scans across multiple IP's can be tracked. Don't  
# remove them from the log yet as we want to see them associated  
# with each host as well.
```

```
grep ' R ' firewall.log > resets.txt
```

```
# The first traffic pattern I noticed in this log was Web server  
# traffic so I first dump it to its own file and then removed it  
# from the working log file. Note the [: ] which tells grep to
```

Firewall Logging & Monitoring

Chris Brenton and Others

```
# match on the character string when it ends with a colon or a
# space character (match '12.33.246.2.80:' or '12.33.247.2.80 ').
# I could not match on just 12.33.247.2.80, because the log also
# contains traffic to 12.33.247.2.8080, thus I need to ensure I
# included the space or colon on the end of the string.
```

```
grep '12.33.247.2.80[: ]' firewall.log > web_server1.txt
grep -v '12.33.247.2.80[: ]' firewall.log > trace1.txt
```

```
# Next pattern I found was DNS so I parse that out. Notice I
# broke up DNS based on transport. This is because TCP DNS is a
# little more interesting as it could be zone transfer attempts or
# queries that had large answers. Also note I did not tile this to
# a single server. All DNS goes into the same file. If you want a
# per server breakdown, use a format similar to the Web server
# example above.
```

```
grep udp trace1.txt | grep '%.53[: ]' > dns_udp.txt
grep tcp trace1.txt | grep '%.53[: ]' > dns_tcp.txt
grep -v '%.53[: ]' trace1.txt > trace2.txt
```

```
# Next pattern found is TCP/4040. Per the Neohapsis services file,
# this is CIPHERim traffic so parse that out.
```

```
grep '12.33.247.2.4040' trace2.txt > CIPHERim_server.txt
grep -v '12.33.247.2.4040' trace2.txt > trace3.txt
```

```
# Next pattern found was SMTP. I decided to look only at the initial
# SYN packet going to the server rather than all traffic. I also
# decided to not limit the target IP as being our server. This will
# permit me to put all SMTP traffic in one file. The only thing bad
# about parsing the data this way is that people probing TCP/25 on
# on non-SMTP servers will get mixed into this file. This will make
# it harder to ID those port scans. If this was a true firewall log
# you could fix that problem by doing a secondary grep on "drop"
# or what ever keyword your firewall uses to say someone hit port 25
# on an IP that is not permitted to communicate on that port.
```

```
grep '%.25: ' trace3.txt | grep ' S ' > smtp.txt
grep -v '%.25[: ]' trace3.txt > trace4.txt
```

```
# Next pattern found was traffic to our proxy server. We've decided to
# save packets in all directions but to limit the save to only traffic
# to and from the legitimate proxy. This way probes to other IP's on
# this port number don't get parsed out so we can catch them further down.
```

```
grep '12.33.247.2.8080' trace4.txt > proxy.txt
grep -v '12.33.247.2.8080' trace4.txt > trace5.txt
```

```
# Next pattern is inbound SSH to our SSH server. Save these off.
```

```
grep '12.33.247.11.22' trace5.txt > ssh_server1.txt
grep -v '12.33.247.11.22' trace5.txt > trace6.txt
```

```
# Looks like we have another SSH server accessible from the Internet.
```

```
grep '12.33.247.10.22' trace6.txt > ssh_server2.txt
grep -v '12.33.247.10.22' trace6.txt > trace7.txt
```

```
# We also have SSH between internal systems as well. Let's put
```

Firewall Logging & Monitoring

Chris Brenton and Others

that in its own file.

```
grep '%.22[: ]' trace7.txt > internal_ssh.txt
grep -v '%.22[: ]' trace7.txt > trace8.txt
```

Looks like we have a second Web server. Save this data off in a similar fashion to the last one except we don't have proxy traffic to worry about.

```
grep '12.33.247.4.80' trace8.txt > web_server2.txt
grep -v '12.33.247.4.80' trace8.txt > trace9.txt
```

Next pattern is RIP routing updates. Note RIP uses a source and destination # port of 123 UDP/520 so we don't need to match multiple patterns

```
grep '%.520: ' trace9.txt > rip_routing.txt
grep -v '%.520: ' trace9.txt > trace10.txt
```

Looks like NTP traffic to sync time is used as well. Let's save this # to its own file.

```
grep '%.123: ' trace10.txt > ntp.txt
grep -v '%.123: ' trace10.txt > trace11.txt
```

Also found SSL traffic. Let's dump that to a file. If we know only # one system should be using SSL we can tie it to a single IP address.

```
grep '12.33.247.8.443' trace11.txt > ssl.txt
grep -v '12.33.247.8.443' trace11.txt > trace12.txt
```

Also found Big Brother used for system monitoring. Let's put that in # it's own file.

```
grep '12.33.247.4.1984' trace12.txt > big-brother.txt
grep -v '12.33.247.4.1984' trace12.txt > trace13.txt
```

Looks like we have another type of proxy on the wire as well. Let's # put that into its own file.

```
grep '%.3307[: ]' trace13.txt > op-session-proxy.txt
grep -v '%.3307[: ]' trace13.txt > trace14.txt
```

Looks like yet another Web server. Save this one off as well.

```
grep '12.33.247.10.80' trace14.txt > web_server3.txt
grep -v '12.33.247.10.80' trace14.txt > trace15.txt
```

ICMP unreachable are interesting as it indicates someone is trying to # get to something they should not or the resource is gone. Let's put # them into their own file so they are easier to review. With this log # file we can just look for "unreachable" but for your firewall you may # need to grep for "icmp" and then pipe it though another grep that flags # type 3 and 11 packets only. Don't want to mix in echo-requests or any # other type of ICMP packets.

```
grep unreachable trace15.txt > icmp-unreachables.txt
grep -v unreachable trace15.txt > trace16.txt
```

Found some RADIUS traffic to parse out.

```
grep '12.33.247.11.1812' trace16.txt > radius.txt
```

Firewall Logging & Monitoring

Chris Brenton and Others

```
grep -v '12.33.247.11.1812' trace16.txt > trace17.txt
```

```
# Let's say at this point we think we are done. Remember back on the first
# line we extracted all the reset packets? Well before we create our file
# with unexpected traffic patterns we need to clean these resets out of the
# working file.
```

```
grep -v ' R ' trace17.txt > trace18.txt
```

```
# Now we just rename the last working file as it will contain all the stuff
# we did not expect to see.
```

```
mv trace18.txt interesting_stuff.txt
```

```
# If you are writing a new script, do not add in the next line till you
# know it works properly as it will make troubleshooting what is broken
# a real pain. Our final line cleans up all those temporary work files
# that get left behind.
```

```
rm -f trace*.txt
```

```
# Grab yesterdays log file, give it a useful name, and change to theworking directory.
```

```
cp /var/log/messages.1 /root/logcheck/full_firewall.log
cd /root/logcheck
```

```
# Resets are an indication of port scans or communication errors# which make them "interesting".
Dump them all in the same file# so that scans across multiple IP's can be tracked. Don't# remove
them from the log yet as we want to see them associated# with each host as well.
```

```
grep ' R ' firewall.log > resets.txt
```

```
# The first traffic pattern I noticed in this log was Web server# traffic so I first dump it to its own file
and then removed it# from the working log file. Note the [: ] which tells grep to# match on the
character string when it ends with a colon or a# space character (match '12.33.246.2.80:' or
'12.33.247.2.80 ').# I could not match on just 12.33.247.2.80, because the log also# contains traffic to
12.33.247.2.8080, thus I need to ensure I# included the space or colon on the end of the string.
```

```
grep '12.33.247.2.80[: ]' firewall.log > web_server1.txtgrep -v '12.33.247.2.80[: ]' firewall.log > trace1.txt
```

```
# Next pattern I found was DNS so I parse that out. Notice I# broke up DNS based on transport. This
is because TCP DNS is a# little more interesting as it could be zone transfer attempts or# queries that
had large answers. Also note I did not tile this to# a single server. All DNS goes into the same file. If
you want a# per server breakdown, use a format similar to the Web server# example above.
```

```
grep udp trace1.txt | grep '%.53[: ]' > dns_udp.txtgrep tcp trace1.txt | grep '%.53[: ]' > dns_tcp.txt
grep -v '%.53[: ]' trace1.txt > trace2.txt
```

```
# Next pattern found is TCP/4040. Per the Neohapsis services file,# this is CIPHERim traffic so parse
that out.
```

```
grep '12.33.247.2.4040' trace2.txt > CIPHERim_server.txtgrep -v '12.33.247.2.4040' trace2.txt > trace3.txt
```

```
# Next pattern found was SMTP. I decided to look only at the initial# SYN packet going to the server
rather than all traffic. I also# decided to not limit the target IP as being our server. This will# permit me
to put all SMTP traffic in one file. The only thing bad# about parsing the data this way is that people
probing TCP/25 on# on non-SMTP servers will get mixed into this file. This will make# it harder to ID
```

Firewall Logging & Monitoring

Chris Brenton and Others

those port scans. If this was a true firewall log# you could fix that problem by doing a secondary grep on "drop"# or what ever keyword your firewall uses to say someone hit port 25# on an IP that is not permitted to communicate on that port.

```
grep '%.25: ' trace3.txt | grep ' S ' > smtp.txtgrep -v '%.25[: ]' trace3.txt > trace4.txt
```

Next pattern found was traffic to our proxy server. We've decided to# save packets in all directions but to limit the save to only traffic# to and from the legitimate proxy. This way probes to other IP's on# this port number don't get parsed out so we can catch them further down.

```
grep '12.33.247.2.8080' trace4.txt > proxy.txtgrep -v '12.33.247.2.8080' trace4.txt > trace5.txt
```

Next pattern is inbound SSH to our SSH server. Save these off.

```
grep '12.33.247.11.22' trace5.txt > ssh_server1.txtgrep -v '12.33.247.11.22' trace5.txt > trace6.txt
```

Looks like we have another SSH server accessible from the Internet.

```
grep '12.33.247.10.22' trace6.txt > ssh_server2.txtgrep -v '12.33.247.10.22' trace6.txt > trace7.txt
```

We also have SSH between internal systems as well. Let's put# that in its own file.

```
grep '%.22[: ]' trace7.txt > internal_ssh.txtgrep -v '%.22[: ]' trace7.txt > trace8.txt
```

Looks like we have a second Web server. Save this data off in a similar # fashion to the last one except we don't have proxy traffic to worry about.

```
grep '12.33.247.4.80' trace8.txt > web_server2.txtgrep -v '12.33.247.4.80' trace8.txt > trace9.txt
```

Next pattern is RIP routing updates. Note RIP uses a source and destination# port of 123 UDP/520 so we don't need to match multiple patterns

```
grep '%.520: ' trace9.txt > rip_routing.txtgrep -v '%.520: ' trace9.txt > trace10.txt
```

Looks like NTP traffic to sync time is used as well. Let's save this# to its own file.

```
grep '%.123: ' trace10.txt > ntp.txtgrep -v '%.123: ' trace10.txt > trace11.txt
```

Also found SSL traffic. Let's dump that to a file. If we know only# one system should be using SSL we can tie it to a single IP address.

```
grep '12.33.247.8.443' trace11.txt > ssl.txtgrep -v '12.33.247.8.443' trace11.txt > trace12.txt
```

Also found Big Brother used for system monitoring. Let's put that in# its own file.

```
grep '12.33.247.4.1984' trace12.txt > big-brother.txtgrep -v '12.33.247.4.1984' trace12.txt > trace13.txt
```

Looks like we have another type of proxy on the wire as well. Let's# put that into its own file.

```
grep '%.3307[: ]' trace13.txt > op-session-proxy.txtgrep -v '%.3307[: ]' trace13.txt > trace14.txt
```

Looks like yet another Web server. Save this one off as well.

```
grep '12.33.247.10.80' trace14.txt > web_server3.txtgrep -v '12.33.247.10.80' trace14.txt > trace15.txt
```

Firewall Logging & Monitoring

Chris Brenton and Others

ICMP unreachable are interesting as it indicates someone is trying to get to something they should not or the resource is gone. Let's put them into their own file so they are easier to review. With this log file we can just look for "unreachable" but for your firewall you may need to grep for "icmp" and then pipe it through another grep that flags type 3 and 11 packets only. Don't want to mix in echo-requests or any other type of ICMP packets.

```
grep unreachable trace15.txt > icmp-unreachables.txtgrep -v unreachable trace15.txt > trace16.txt
```

Found some RADIUS traffic to parse out.

```
grep '12.33.247.11.1812' trace16.txt > radius.txtgrep -v '12.33.247.11.1812' trace16.txt > trace17.txt
```

Let's say at this point we think we are done. Remember back on the first line we extracted all the reset packets? Well before we create our file with unexpected traffic patterns we need to clean these resets out of the working file.

```
grep -v ' R ' trace17.txt > trace18.txt
```

Now we just rename the last working file as it will contain all the stuff we did not expect to see.

```
mv trace18.txt interesting_stuff.txt
```

If you are writing a new script, do not add in the next line till you know it works properly as it will make troubleshooting what is broken a real pain. Our final line cleans up all those temporary work files that get left behind.

```
rm -f trace*.txt
```