MIKE RASH

IDS signature matching with iptables, psad, and fwsnort

Michael Rash holds a Master's degree in Applied Mathematics and works as a Security Architect for Enterasys Networks, Inc. He is the creator of the cipherdyne.org suite of open source security tools and is author of the book *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort,* published by No Starch Press.

mbr@cipherdyne.org

THE ANALYSIS OF LOG DATA IS BECOMing an increasingly important capability as more applications generate copious amounts of run-time information. This information often has interesting things to say for those who are listening (including evidence of events that are significant from a security perspective), but the sheer volume of information often requires automated tools to make sense of the data. The iptables firewall is built on top of the Netfilter framework in the Linux kernel, and it includes the ability to create verbose syslog messages of the network and transport layer headers associated with IP packets. In addition, through the use of the iptables string match extension, the application layer can be searched for evidence of malicious activity and iptables can then log or take action against such packets.

This article explores the use of psad and fwsnort [1] to automate the analysis of iptables log messages with a particular emphasis on passive OS fingerprinting and the detection of application-layer attacks. Both psad and fwsnort are open-source software released under the GNU Public License (GPL). Some familiarity with iptables and the Snort rules language is assumed in this article [2]. Also, see the INSTALL file bundled with the psad and fwsnort sources for installation instructions.

Network Setup and Default iptables Policy

I will illustrate network traffic against a Linux system that is protecting a small internal network with an iptables policy that implements a default "log and drop" stance for any traffic that is not necessary for basic connectivity. In particular, the iptables policy provides NAT services to allow clients on the internal network to issue DNS and Web requests out through the firewall (with the internal network having the RFC 1918 subnet 192.168.10.0/24 and the external interface on the firewall having a routable IP address), and the firewall accepts SSH connections from the internal network. The iptables policy uses the Netfilter connection tracking capability to allow traffic associated with an established TCP connection to pass through; also allowed are packets that are responses to UDP datagrams (which may include ICMP

port unreachable messages in response to a UDP datagram to a port where no server is bound). All other traffic is logged and dropped (with iptables log messages reported via the kernel logging daemon klogd to syslog). This iptables policy is implemented by the following iptables commands [3]:

```
iptables -F INPUT
iptables -P INPUT DROP
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -i eth1 -p tcp -s 192.168.10.0/24 --dport 22 \
   -m state --state NEW -j ACCEPT
iptables -A INPUT -i ! lo -j LOG --log-ip-options \
   --log-tcp-options --log-prefix "DROP "

iptables -F FORWARD
iptables -P FORWARD DROP
iptables -A FORWARD -m state --state ESTABLISHED,RELATED -j ACCEPT
iptables -A FORWARD -p tcp -s 192.168.10.0/24 --dport 80 -m state \
   --state NEW -j ACCEPT
iptables -A FORWARD -p tcp -s 192.168.10.0/24 --dport 443 -m state \
   --state NEW -j ACCEPT
iptables -A FORWARD -p udp -s 192.168.10.0/24 --dport 53 -j ACCEPT
iptables -A FORWARD -i ! lo -j LOG --log-ip-options \
   --log-tcp-options --log-prefix "DROP "

iptables -t nat -A POSTROUTING -o eth0 -s 192.168.10.0/24 \
   -j MASQUERADE
```

Passive OS Fingerprinting

With the iptables policy active on the Linux system, it is time to see what it can show us from a logs perspective. First, from a system on the internal network (with hostname "int" and IP address 192.168.10.50), we attempt to initiate a TCP connection to port 5001 on the firewall (where the firewall's internal IP is 192.168.10.1):

```
[int]$ nc 192.168.10.1 5001
```

This results in the following iptables log message for the incoming TCP SYN packet, which is blocked and logged by iptables:

```
Sep 13 21:22:24 fw kernel: DROP IN=eth1 OUT= \
MAC=00:13:46:3a:41:4b:00:0c:41:24:56:37:08:00 \
SRC=192.168.10.50 DST=192.168.10.1 LEN=60 TOS=0x00 PREC=0x00 \
   TTL=64 ID=51104 DF PROTO=TCP SPT=57621 DPT=5001 \
WINDOW=5840 RES=0x00 SYN URGP=0 \
OPT (020405B40402080A1ECB4C4C0000000001030302)
```

The log message contains, among other things, source and destination IP addresses, the IP ID and TTL values, source and destination port numbers, TCP flags (with just the SYN flag being set in this case), and the options portion of the TCP header (preceded by the "OPT" string). From the perspective of passively fingerprinting the operating system that generated the TCP SYN packet against the firewall, the most interesting fields in the log message are as follows:

- IP length: LEN=60
- TTL: TTL=64
- The Don't Fragment bit: DF
- TCP window size: WINDOW=5840
- TCP flags: SYN
- TCP options: OPT (020405B40402080A003F83040000000001030302)

(Note that the TCP options string is only included within an iptables log message if the --log-tcp-options argument is given on the iptables command line when adding the LOG rule.) These fields are important because they are the same fields that the best-known passive OS fingerprinting software, p0f, uses to fingerprint operating systems [4]. This illustrates the completeness of the iptables logging format, because it is possible to implement the same passive OS fingerprinting algorithm used by p0f but use iptables log messages as input instead of sniffing packet data off the wire with a packet capture library. The psad project implements the p0f fingerprinting algorithm over iptables log messages, and the TCP log message just listed conforms to the following p0f fingerprint:

```
S4:64:1:60:M*,S,T,N,W2:   Linux:2.5::Linux 2.5 (sometimes 2.4)
```

This fingerprint specifies a series of requirements on packet headers separated by colons and is read as follows:

- "S4" requires that the TCP window size be four times as large as the Maximum Segment Size (MSS). The MSS value is part of the TCP options field.
- "64" matches the TTL value and requires that the initial TTL is 64. (This value has to be estimated for packets that traverse the open Internet.)
- "1" requires that the Don't Fragment bit is set.
- "60" requires that the overall size of the SYN packet (including the IP header) be 60 bytes.
- "M*,S,T,N,W2" describes the options field of the TCP header; "M*" means any MSS size, "S" means Selective Acknowledgment is OK, "T" means that the TCP options contain a time stamp, "N" requires a NOP option, and "W2" requires a window scaling value of 2.

Decoding the options string from the iptables log message is the most complex portion of the fingerprinting activity. The options string follows Type Length Value (TLV) encoding, where each TCP option has one byte for the option type, one byte for the option length, and a variable number of bytes for the option value [5]. Hence, the options string decodes to the following, which matches the requirements of the "Linux:2.5::Linux 2.5" p0f signature (and psad reports this fingerprint within email alerts that it generates [6]):

```
- MSS: 1460
- Selective Acknowledgment is OK
- Timestamp: 516639820
- NOP
- Window scaling value: 2
```

Snort Rule Matching with fwsnort

In the previous section, we saw that it is possible to collect iptables log messages for SYN packets sent from arbitrary hosts and, in many cases, infer the OS that generated these packets. Passively fingerprinting operating systems is a nice trick and can reveal interesting information about an attacker, but in the threat environment on the Internet today the real action is at the application layer (OS fingerprinting only requires the inspection of network and transport layer headers). To get a feel for how important application-layer inspection is to computer security, one need only examine the Snort rule set. In Snort version 2.3.3 (the last version of Snort that included rules released under the GPL instead of the VRT service from Sourcefire), there are about 150 signatures out of 3,000 that only test packet headers and have no application-layer match requirement.

In the Snort rules language, elements that test the application layer include the "content," "uricontent," "pcre," "byte_test," "byte_jump," and "asn1" keywords, whereas elements such as "flags," "ack," "seq," and "ipopts" (among others) test packet header fields. Maintaining an effective intrusion detection stance for network traffic requires the ability to inspect application-layer data, and 95% of all Snort rules are focused on the application layer.

The fwsnort project translates Snort rules into iptables rules that are designed to detect (and optionally react to) the same attacks, and the Snort 2.3.3 rule set is packaged with fwsnort. Because the detection capabilities of iptables are limited to matches on strings via the string match extension [7], many Snort rules (such as those that contain a pcre match) cannot be translated. Still, about 60% of all Snort 2.3.3 rules can be translated into iptables rules by fwsnort because iptables provides a flexible set of facilities to the user for matching traffic in kernel space. Chief among these facilities is the ability to match on multiple content strings instead of just a single string; iptables 1.3.6 introduced this capability by allowing multiple matches of the same type to be specified on the iptables command line. In the following we will see an example of a Snort rule that looks for two malicious content strings returned from a Web server and will see how iptables can be made to look for the same attack in network traffic.

Some of the most interesting and devastating attacks today exploit vulnerabilities in client applications that are attacked via malicious or compromised servers. Because in many cases thousands of independent client applications communicate with popular servers, an attacker can take advantage of this multiplying effect just by compromising a heavily utilized server and forcing it to launch attacks against any hapless client who connects to it.

An example of a Snort rule that looks for a client-side attack against a Web browser is rule ID 1735, which is labeled as "WEB-CLIENT XMLHttpRequest attempt." This rule detects a possible attempt to force a Web browser to return a list of files and directories on the system running the browser back to the attacker, via the responseText property, after redirecting the browser to point to the local filesystem. Fortunately, this attack applies to older versions of the Netscape and Mozilla browsers, but if a Web server sends data that matches this Snort rule back to a Web browser running on my network, I would want to know about it regardless of whether or not the browser is vulnerable. The XMLHttpRequest attack is tracked in the Common Vulnerabilities and Exposures (CVE) database as CVE-2002-0354 [8]. Here is the Snort rule for this attack:

```
alert tcp $EXTERNAL_NET $HTTP_PORTS -> $HOME_NET any \
(msg:"WEB-CLIENT XMLHttpRequest attempt"; \
flow:to_client,established; content:"new XMLHttpRequest|28|"; \
content:"file|3A|//"; nocase; reference:bugtraq,4628; \
reference:cve,2002-0354; classtype:web-application-attack; \
sid:1735; rev:7;)
```

Note that the Snort rule is looking for two content strings that emanate from an external Web server (with the source IP being $EXTERNAL_NET and the source port being $HTTP_PORTS) back to a Web client that is on the internal network (with the destination IP being $HOME_NET and the destination port being "any," since the local TCP stack would choose a random high port for the Web session). The two content strings are "new XMLHttpRequest|28|" and "file|3A|//". Each of these strings specifies one byte by its hex code between pipe characters: "|28|" in the first content string, and "|3A|" in the second. So, when translating this Snort rule into an iptables rule, we must account for that. With fwsnort installed, let's use it to translate

Snort rule ID 1735 and then load it into the iptables policy on the firewall (some output below has been abbreviated):

```
[fw]# fwsnort --snort-sid 1735
[+] Parsing Snort rules files...
[+] Found sid: 1735 in web-client.rules
[+] iptables script: /etc/fwsnort/fwsnort.sh
[fw]#  /etc/fwsnort/fwsnort.sh
[+] Adding web-client rules.
```

Examine the /etc/fwsnort/fwsnort.sh script and you can see the iptables command below. This command uses the --hex-string argument so that the Snort content fields can be specified "as is" within the iptables command (with the bytes between the pipe characters being properly interpreted), and the rule target instructs iptables to log any matching packet with the prefix "[1] SID1735 ESTAB". This prefix informs the user that Snort rule ID 1735 was detected within an established TCP connection (fwsnort interfaces with the Netfilter connection tracking capability for this), and the rule is the first rule "[1]" within the FWSNORT_FORWARD_ESTAB chain.

The "--algo bm" argument instructs the string match extension to use the Boyer-Moore string-matching algorithm to conduct the application-layer match. With kernels in the Linux 2.6 series, the string match extension leverages a text-matching infrastructure implemented in the kernel which supports multiple string-matching algorithms; the Boyer-Moore algorithm exhibits excellent performance characteristics and is commonly used within open source and proprietary intrusion detection systems. Finally, the iptables comment match is used to include the Snort rule "msg," "classtype," and "reference" fields within the iptables rule for easy viewing under a command such as "iptables -v -n -L FWSNORT_FORWARD_ESTAB." We then have:

```
$IPTABLES -A FWSNORT_FORWARD_ESTAB -d 192.168.10.0/24 -p tcp \
--sport 80 -m string --hex-string "new XMLHttpRequest|28|" \
--algo bm -m string --hex-string "file|3A|//" --algo bm -m comment \
--comment "sid:1735; msg:WEB-CLIENT XMLHttpRequest attempt;
classtype:web-application-attack; reference:bugtraq,4628; rev:7; \
FWS:1.0.1;" -j LOG --log-ip-options --log-tcp-options --log-prefix \
"[1] SID1735 ESTAB "
```

Now let us simulate the XMLHttpRequest attack through the iptables firewall against an internal Web browser. For this, we use Perl and Netcat on a dummy Web server at IP 11.11.1.1 (a randomly selected IP address for illustration purposes only). The following Perl command sends data matching the two content fields in Snort rule ID 1735 back to the Web client as soon as it connects:

```
[webserver]# perl -e 'printf "new XMLHttpRequest\x28AAAAAAAfile\x3A//"' |nc -l -p 80
[int]$ nc -v 11.11.1.1 80
Connection to 11.11.1.1 80 port [tcp/www] succeeded!
new XMLHttpRequest(AAAAAAAfile://
```

The last line here shows that the Web client received data that matches the Snort rule; iptables has not interfered with the traffic and has happily let it pass into the internal network. On the firewall, we see the following iptables log message (note that the "[1] SID1735 ESTAB" log prefix and the ACK and PSH flags are set, since this packet was matched within an established TCP connection):

```
Sep 14 08:39:24 fw kernel: [1] SID1735 ESTAB IN=eth0 OUT=eth1 \
SRC=11.11.1.1 DST=192.168.10.50 LEN=85 TOS=0x00 PREC=0x00 TTL=63 \
```

```
 ID=23507 DF PROTO=TCP SPT=80 DPT=34646 WINDOW=91 RES=0x00 ACK PSH \
URGP=0 OPT (0101080A650A550A1F663D7A)
```

At this point we are confident that iptables is able to detect the attack. However, because iptables is a firewall, it is also inline to the traffic whereas Snort (unless deployed in inline mode) is merely able to passively monitor the traffic. Let us take advantage of this by changing the fwsnort command. This time we use the --ipt-reject command-line argument to have fwsnort use the iptables REJECT target against the Web connection in order to knock it down with a TCP RST packet:

```
[fw]# fwsnort --snort-sid 1735 --ipt-reject
[+] Parsing Snort rules files...
[+] Found sid: 1735 in web-client.rules
[+] iptables script: /etc/fwsnort/fwsnort.sh
[fw]#  /etc/fwsnort/fwsnort.sh
[+] Adding web-client rules.
```

Let us run the attack simulation once more:

```
[webserver]# perl -e 'printf "new \
   XMLHttpRequest\x28AAAAAAAfile\x3A//"' |nc -l -p 80
[int]$ nc -v 11.11.1.1 80
Connection to 11.11.1.1 80 port [tcp/www] succeeded!
```

We see that the client is again able to successfully establish a TCP connection with the Web server (that is, the TCP three-way handshake is allowed to complete), but no data comes across. This is because of the TCP RST generated by the REJECT target against the Web server. The REJECT target only sends the RST to the IP address that triggered the rule match within iptables, so the Web client never sees it. However, the iptables REJECT target is a terminating target, so it also drops the matching packet (in this case the packet that contains the XMLHttpRequest string). Hence, the malicious traffic never makes it to the targeted TCP stack, and this is an important capability when some attacks only require a single packet in order to do their dirty work (the SQL Slammer worm is a good example). Only an inline device can prevent individual malicious packets from reaching their intended target.

On the firewall, fwsnort has also created a logging rule that produces the following log message (note that the log prefix now includes the string "REJ", indicating that the packet was rejected):

```
Sep 14 08:41:24 fw kernel: [1] REJ SID1735 ESTAB IN=eth0 OUT=eth1 \
SRC=11.11.1.1 DST=192.168.10.50 LEN=85 TOS=0x00 PREC=0x00 TTL=63 \
  ID=46352 DF PROTO=TCP SPT=80 DPT=52078 WINDOW=91 RES=0x00 ACK PSH \
URGP=0 OPT (0101080A650ACA031F66B26C)
```

Conclusion

This article has focused on two relatively advanced usages of functionality provided by the iptables firewall: the completeness of the log format, which makes passive OS fingerprinting possible, and the ability to inspect application-layer data for evidence of malicious activity. The psad and fwsnort projects automate both of these tasks and can provide an important additional security layer to an iptables firewall. The Snort community has guided the way to effective attack detection on the Internet today, and iptables can leverage the power of this community to extend a filtering policy into the realm of application inspection. Armed with such a policy, iptables becomes a sentry against application-layer attacks.

REFERENCES

[1] http://www.cipherdyne.org/.

[2] Snort rule writing documentation: http://www.snort.org/docs/writing_rules/chap2.html.

[3] A script that implements the default iptables policy can be downloaded from http://www.cipherdyne.org/LinuxFirewalls/ch01/iptables.sh.tar.gz.

[4] Passive OS fingerprinting is really passive stack fingerprinting. That is, the IP and TCP stacks used by various operating systems exhibit slight differences, and detecting these differences (i.e., "fingerprinting" the stack) can allow the operating system that uses the stack to be guessed; see http://lcamtuf.coredump.cx/p0f.shtml.

[5] There are two exceptions to this: The "NOP" and "End of Option List" options are only one byte long. See RFC 793 for more information.

[6] See http://www.cipherdyne.org/psad/docs for examples of such alerts.

[7] The iptables u32 extension is being reintegrated with the Netfilter framework after it was deprecated late in the 2.4 series kernel, so more complicated arithmetic tests can be written against both packet headers and application-layer data. The u32 extension essentially emulates the "byte_test" operator in the Snort rules language.

[8] See http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0354.