

## Packet Crafting for Firewall & IDS Audits (Part 2 of 2)

Don Parker 2004-07-19

### Introduction

This is the second of a two-part article series that discusses various methods of testing the integrity of your firewall and IDS, using low-level TCP/IP packet crafting tools and techniques. [Part one](#) showed several examples that tested a firewall (port 80 TCP, and port 53, UDP) using tools like *hping* and *tcpdump*.

We will now continue the discussion with a third test of the firewall, using the same tools as noted above, and then move on to test your IDS signatures and detection ability. Note that the focus here is on a Linux environment, but the process is similar with other Unix-like firewall/IDS environments as well.

Please familiarize yourself with [part one](#) of this article before continuing on with this paper.

### Testing your firewall - third example, ICMP echo requests

The example shown below is a simple ICMP echo request to see if a machine is alive, in this case our test machine.

As was used extensively in part one of this article, we'll be using *hping* and *tcpdump* to perform the tests. The syntax and output of *hping* for a simple ICMP echo request is shown below:

```
monkeylabs:/home/don # hping -l 192.168.1.108 -c 1
HPING 192.168.1.108 (eth0 192.168.1.108): icmp mode set, 28 headers + 0 data bytes
len=46 ip=192.168.1.108 ttl=64 id=122 icmp_seq=0 rtt=0.4 ms

--- 192.168.1.108 hping statistic ---
1 packets tramitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
```

This time there is one packet received as was noted above, and the round trip time is shown. By going with this it looks like our ICMP echo request was indeed allowed through the firewall.

Below you will note the packet as it was sent on our *hping* machine. The output shows the packet was received on our *hping* test machine from the machine we just pinged from. The *tcpdump* syntax is shown below once again.

```

monkeylabs:/home/don # tcpdump -nXvs 0 ip and host 192.168.1.100 and host
192.168.1.108
tcpdump: listening on eth0

11:10:37.458058 192.168.1.100 > 192.168.1.108: icmp: echo request (ttl 64, id 51585,
len 28)
0x0000   4500 001c c981 0000 4001 2d3f c0a8 0164           E.....@.-?...d
0x0010   c0a8 016c 0800 5dd0 9a2f 0000                   ...l..]../..

11:10:37.458260 192.168.1.108 > 192.168.1.100: icmp: echo reply (ttl 64, id 117, len
28)
0x0000   4500 001c 0075 0000 4001 f64b c0a8 016c           E....u..@..K...l
0x0010   c0a8 0164 0000 65d0 9a2f 0000 0000 0000           ...d..e../.....
0x0020   0000 0000 0000 0000 0000 0000 0000 0000           .....

```

We can deduce from the above two packets that the destination computer's firewall did allow the ICMP packet through, as we received an answer packet back from it. I did not show the destination computer receiving the packet as we have successfully proven that the firewall is accepting our test criteria. Nor is there any firewall output to show as it did not cause any firewall rulesets to trigger.

## IDS signature test - reserved flag test

We will now begin to do some testing of our IDS, which in this case is Snort. This test was done with Snort's default ruleset.

The first test that we will do is to see if our IDS triggers as it is supposed to when it receives packets with the ECN and CWR flags active in the 13th byte offset of the TCP header.

*Hping* syntax is as noted below:

```

monkeylabs:/home/don # hping -X -Y 192.168.1.108 -p 80 -c 1
HPING 192.168.1.108 (eth0 192.168.1.108): XY set, 40 headers + 0 data bytes

--- 192.168.1.108 hping statistic ---
1 packets tramitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms

```

Noted below is the packet as seen when it was sent by the machine which had crafted it:

```

/home/don # tcpdump -nXvs 0 tcp and host 192.168.1.100 and 192.168.1.108
tcpdump: listening on eth0

08:42:00.081599 192.168.1.100.1215 > 192.168.1.108.80: WE [tcp sum ok] win 512 (ttl
64, id 29279, len 40)
0x0000  4500 0028 725f 0000 4006 8450 c0a8 0164      E..(r_..@..P...d
0x0010  c0a8 016c 04bf 0050 460e ae8b 6c89 fe96      ...l...PF...l...
0x0020  50c0 0200 c43a 0000                          P....:..

```

Note the flags WE in the above packet. These signify that the ECN and CWR flags are set in this packet.

Now, note the packet below comes from the IDS test machine.

```

/home/don # tcpdump -nXvs 0 tcp and host 192.168.1.108 and 192.168.1.100
tcpdump: listening on eth0

08:40:58.589496 192.168.1.100.1215 > 192.168.1.108.80: WE [tcp sum ok] win
512
(ttl 64, id 29279, len 40)
0x0000  4500 0028 725f 0000 4006 8450 c0a8 0164      E..(r_..@..P...d
0x0010  c0a8 016c 04bf 0050 460e ae8b 6c89 fe96      ...l...PF...l...
0x0020  50c0 0200 c43a 0000 0000 0000 0000          P....:.....

```

The packet is as it should be: a mirror image of the packet sent by the machine on which it was crafted.

Now let's take a look at the Snort output. Below you will note the Snort output shown as it received the packet. You will see that there is a number 1 and 2 that are set in the output. This signifies that the proper bits for this byte are set. In this case it is bit 128 and 64 in the 13th byte offset in the TCP header. This is where all the flags are contained, in other words, flags such as SYN, FYN, PSH and such.

There is a great deal of information logged in the Snort output. Starting on the first line and going from left to right I will explain the fields themselves. We start off with the date and time field, down to the microsecond (just like *tcpdump*). This is followed by the MAC addresses of the sending and receiving computers. Next is the type field which stands for DoD ethernet, and then the length of the packet itself. Now the source IP address and port is followed by the destination IP address and port. After that, you see TCP which is for the TCP packet and a time to live of 64. The type of service is shown to be zero. Then the IP id number is shown as 29279, and after it is the IP header length of 20 bytes. DgmLen stands for the datagram length. As mentioned above, the 1 and 2 mean that the bit positions 128 and 64 are set in this packet. Then we have the sequence number, as well as the acknowledgement number. Closing out this info is the window size and TCP length.

Here is the Snort output from our first example:

```
03/03-08:40:58.589496 0:C:6E:8C:D4:61 -> 0:50:DA:C5:9D:8B type:0x800 len:0x3C
192.168.1.100:1215 -> 192.168.1.108:80 TCP TTL:64 TOS:0x0 ID:29279 IpLen:20
DgmLen:40
12***** Seq: 0x460EAE8B Ack: 0x6C89FE96 Win: 0x200 TcpLen: 20
```

In the `/var/log/snort/alert` file, the output noted below was logged as a result of the signature this packet triggered when it was parsed by Snort:

```
linux:/var/log/snort # more alert
[**] [111:1:1] (spp_stream4) STEALTH ACTIVITY (unknown) detection [**]
03/03-08:40:58.589496 0:C:6E:8C:D4:61 -> 0:50:DA:C5:9D:8B type:0x800 len:0x3C
192.168.1.100:1215 -> 192.168.1.108:80 TCP TTL:64 TOS:0x0 ID:29279 IpLen:20
DgmLen:40
12***** Seq: 0x460EAE8B Ack: 0x6C89FE96 Win: 0x200 TcpLen: 20
```

Our example was used to see if the Snort IDS would indeed trigger and log this as an alert, on receipt of a packet with the ECN and CWR flags set. As noted, it did indeed perform as it should have.

## Second IDS signature test - LSRR packets

We will now test a different IDS signature. To be specific, we will see if Snort will indeed detect LSRR ([Loose Source Record Route](#), commonly known as loose source routed) packets as it is supposed to.

*Hping* syntax as noted is below:

```
monkeylabs:/home/don # hping -S --lsrr 192.168.1.108 192.168.1.102 -p 25 -c 1
HPING 192.168.1.102 (eth0 192.168.1.102): S set, 40 headers + 0 data bytes

--- 192.168.1.102 hping statistic ---
1 packets tramitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
```

The packets as they looked like when leaving the hping machine

```

monkeylabs:/home/don # tcpdump -nXvs 0 ip and host 192.168.1.100 and 192.168.1.108
tcpdump: listening on eth0

09:23:35.134313 192.168.1.100.1636 > 192.168.1.108.25: S [tcp sum ok]
384787509:384787509(0) win 512 (ttl 64, id 57275, len 48, optlen=8 LSRR
{192.168.1.102#} EOL)
0x0000  4700 0030 dfbb 0000 4006 7b22 c0a8 0164          G..0....@.{"...d
0x0010  c0a8 016c 8307 08c0 a801 6600 0664 0019          ...l.....f..d..
0x0020  16ef 6435 3ac2 97be 5002 0200 d59f 0000          ..d5:...P.....

```

The Snort output below is shown as Snort saw the packet:

```

03/03-09:22:33.600331 0:C:6E:8C:D4:61 -> 0:50:DA:C5:9D:8B type:0x800 len:0x3E
192.168.1.100:1636 -> 192.168.1.108:25 TCP TTL:64 TOS:0x0 ID:57275 IpLen:28
DgmLen:48
IP Options (1) => LSRR
*****S* Seq: 0x16EF6435  Ack: 0x3AC297BE  Win: 0x200  TcpLen: 20

```

Taken from the alert file in `/var/log/snort/`, we see the output below:

```

[**] [1:501:2] MISC source route lssre [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
03/03-09:22:33.600331 0:C:6E:8C:D4:61 -> 0:50:DA:C5:9D:8B type:0x800 len:0x3E
192.168.1.100:1636 -> 192.168.1.108:25 TCP TTL:64 TOS:0x0 ID:57275 IpLen:28
DgmLen:48
IP Options (1) => LSRR
*****S* Seq: 0x16EF6435  Ack: 0x3AC297BE  Win: 0x200  TcpLen: 20
[Xref => http://www.whitehats.com/info/IDS420][Xref => http://cve.mitre.org/
cgi-bin/cvename.cgi?name=CVE-1999-0909][Xref =>
http://www.securityfocus.com/bid/646]

```

And here is what was received on the test machine:

```
linux:/home/don # tcpdump -nXvs 0 ip and host 192.168.1.108 and 192.168.1.100
tcpdump: listening on eth0
09:22:33.600331 192.168.1.100.1636 > 192.168.1.108.25: S [tcp sum ok]
384787509:384787509(0) win 512 (ttl 64, id 57275, len 48, optlen=8
LSRR{192.168.1.102#} EOL)
0x0000  4700 0030 dfbb 0000 4006 7b22 c0a8 0164      G..0....@.{"...d
0x0010  c0a8 016c 8307 08c0 a801 6600 0664 0019      ...l.....f..d..
0x0020  16ef 6435 3ac2 97be 5002 0200 d59f 0000      ..d5:...P.....
```

So as we can see, the Snort IDS definitely picked up the receipt of our LSRR packets. Now we can be confident that even Snort's default ruleset is working as advertised.

## Dispelling some myths

Finally, I would like to take the time to dispel some myths here about packet crafting. The first myth is about buffer overflows. You cannot send buffer overflows using a packet crafting tool because the three-way TCP/IP handshake needs to be completed for an exploit to work. This is why the code needs to be compiled, for within the code itself are the systems calls to handle the 3 way TCP/IP handshake.

On most stacks today the sequence numbers are pseudo-random and therefore difficult to predict, making the ISN prediction attack impractical, though not impossible. So in light of this any data you inject into a packet will be ignored by the destination machine unless you have successfully completed the TCP/IP handshake.

## Conclusion

Hopefully you can now see the clear benefits of crafting packets, and what it will bring to you and your security posture. Not only that, but along the way you will also learn about the all-important subject of low level TCP/IP. The examples that were shown in this, and the [first part](#) of this article, are just a starting point -- each person's network or setup can be unique, and the firewall rulesets or IDS signatures will be different. It will be up to you to use the knowledge gained to actively probe your own defenses.

I sincerely hope you found this article series of use to you. Feel free to contact me, if you like, with regards to it.

### About the author

[Don Parker](#) is an Intrusion Detection Specialist who holds the GCIA certification. He works for Rigel Kent Security and Advisory Services as an instructor and also provides other computer security services of a highly specialized nature.

View [more articles](#) by Don Parker on SecurityFocus.

*Comments or reprint requests can be sent to the [editor](#).*

[Privacy Statement](#)

Copyright 2006, SecurityFocus