Inline Normalization with Snort 2.9.0

Russ Combs

Snort 2.9.0 can take a more active role in securing your network in inline deployments by normalizing packets and streams to minimize the chance that Snort incorrectly models end systems.

To accomplish this, a new preprocessor was added. You must configure with this option to build it:

```
./configure --enable-normalizer
```

Then you can update your `snort.conf` as follows:

```
config min_ttl: <ttl>
  config new_ttl: <ttl>

  preprocessor normalize_ip4: [df], [rf]
  preprocessor normalize_icmp4
  preprocessor normalize_ip6
  preprocessor normalize_icmp6
  preprocessor normalize_tcp: \
       [ips] [urp] \
       [ecn <ecn_type>], \
       [opts [allow <allowed_opt>+]]
```

For details on these normalizations, see `README.normalize`. We will see the `ttl` and ips normalizations in action below.

Demo Setup

You can run these tests in readback mode using the dump DAQ or in playback mode using tcpreplay and an inline sensor. Using a sensor is the ultimate but you may find the dump DAQ to be indispensable for pcap testing.

Either way, we will run the tests twice, once in IDS mode and again in IPS mode. This will allow us to compare the results.

To run these tests you will need this tarball: http://labs.snort.org/files/normalize.tgz

For readback mode you will need:

1. A system with the latest Snort and DAQ binaries. Install the tarball from this post there.

cd into the Sensor directory and edit `config.sh` to set SNORT appropriately.

When you run the tests with `./readback.sh, inline-out.pcap` will be created which you can examine with wireshark. These files will have all packets as they would appear on the wire exiting Snort.

For playback mode you will need:

1. A source / attack system with the Source/ directory from the tarball. This system also needs tcpreplay.

2. An inline Snort sensor with the Sensor/ directory from the tarball. This system also needs the latest Snort and DAQ binaries.

3. A sink / target system with the Sink/ directory from the tarball. This system also needs tcpdump.

Patch your Source and Sink to opposite sides of the inline pair on your Sensor and edit `config.sh` on each system to configure the PORT_* variables. On your sensor, you will also need to indicate where your Snort and any dynamic DAQ modules are installed. (If you built static DAQs the latter is not required.) You can add any extra options for Snort to SNORT.

```
+--------+      +--------+      +------+
| Source |S>---A>| Sensor |B>---R>| Sink |
+--------+      +--------+      +------+
```

The `inline.sh` script on the sensor will use the `afpacket` DAQ because it requires minimal external configuration. If you want to use a different DAQ, you must change this script and configure your system accordingly. Fore more on the DAQ, see this post.

NOTE: For these tests we are using canned traffic that is replayed from a packet capture file (PCAP file). For the stream tests, both ends of the traffic will be sent from the same point. Although that would not be the case for an inline sensor with live traffic, Snort will still see the traffic in the same relative order it would see live traffic.

Packet Normalization

This test will demonstrate how packets can be modified to meet the needs of a secure network. By ensuring that the time-to-live (TTL) field has a suitable value, it is less likely that a packet will be dropped by the network after being processed by Snort. This enables Snort to more accurately model the receiving host system and prevents wasting time on detection of packets that won't reach their destination. Note that for IP6 the situation is essentially the same, except the equivalent field is called ‚Äúhop limit‚Äù.

TTL normalization pertains to both IP4 TTL (time-to-live) and IP6 (hop limit) and is only performed if both the relevant protocol normalization is enabled and the minimum and new TTL values are configured, as follows:

```
    preprocessor normalize_ip4
    preprocessor normalize_ip6

    config min_ttl: 5
    config new_ttl: 8
```

If a packet is received with a TTL < 5, the TTL will be set to 8. Since we enabled IP6 normalization, the same applies to hop limits.

Readback:
0. From Sensor/ run ./readback.sh ttl_i?s.conf ../Source/ttl.pcap.

Playback:

1. On the sensor, run ./inline.sh ttl_i?s.conf.
2. On the sink, run ../recv.sh.
3. On the source, run ./send.sh ttl.pcap.
4. Type Ctl-C on the sensor and sink to terminate.

The results for ttl_ids.conf are:

```
16:21:10.133823 10.1.2.3.48620 > 10.9.8.7.8: [udp sum ok] udp 16 (ttl 6, id 1, len 44)
0x0000   4500 002c 0001 0000 0611 96ad 0a01 0203        E..,............
0x0010   0a09 0807 bdec 0008 0018 97a3 4352 4f57        ............CROW
0x0020   443a 2020 4120 7769 7463 6821 0000             D:..A.witch!..

16:21:10.133824 10.1.2.3.48620 > 10.9.8.7.8: [udp sum ok] udp 16 (ttl 5, id 2, len 44)
0x0000   4500 002c 0002 0000 0511 97ac 0a01 0203        E..,............
0x0010   0a09 0807 bdec 0008 0018 95dd 2020 4120        ..............A.
0x0020   7769 7463 6821 2020 4120 7769 0000             witch!..A.wi..

16:21:10.133826 10.1.2.3.48620 > 10.9.8.7.8: [udp sum ok] udp 16 (ttl 4, id 3, len 44)
0x0000   4500 002c 0003 0000 0411 98ab 0a01 0203        E..,............
0x0010   0a09 0807 bdec 0008 0018 6785 7463 6821        ..........g.tch!
0x0020   2020 5765 2776 6520 676f 7420 0000             ..We've.got...
```

The results for `ttl_ips.conf` are:

```
0x0000   4500 002c 0001 0000 0611 96ad 0a01 0203        E..,............
0x0010   0a09 0807 bdec 0008 0018 97a3 4352 4f57        ............CROW
0x0020   443a 2020 4120 7769 7463 6821 0000             D:..A.witch!..

16:09:56.871043 10.1.2.3.48620 > 10.9.8.7.8: [udp sum ok] udp 16 (ttl 5, id 2, len 44)
0x0000   4500 002c 0002 0000 0511 97ac 0a01 0203        E..,............
0x0010   0a09 0807 bdec 0008 0018 95dd 2020 4120        ..............A.
0x0020   7769 7463 6821 2020 4120 7769 0000             witch!..A.wi..

16:09:56.871044 10.1.2.3.48620 > 10.9.8.7.8: [udp sum ok] udp 16 (ttl 8, id 3, len 44)
0x0000   4500 002c 0003 0000 0811 94ab 0a01 0203        E..,............
```

```
0x0010   0a09 0807 bdec 0008 0018 6785 7463 6821        ..........g.tch!
0x0020   2020 5765 2776 6520 676f 7420 0000             ..We've.got...
```

In this case you see that the TTL of the third packet was changed from 4 to 8. Since the minimum allowed TTL was configured to 5, there was no change to the first two packets.

```
Packet I/O Totals:
   Received:           57
   Analyzed:            3 (  5.263%)
   Filtered:           54 ( 94.737%)

Verdicts:
      Allow:            2 (  3.509%)
    Replace:            1 (  1.754%)

Normalizer statistics:
               ip4::ttl: 1
```

The ids counts above show 3 analyzed, 2 allowed, and 1 replaced, which is the packet counted under normalizer statistics.

Stream Normalization

This test demonstrates how TCP payload data can be normalized to ensure consistency with retransmitted data. By doing so, Snort is better protected against the vagaries of host dependent TCP reassembly procedures and is less likely to be evaded by such scenarios.

TCP normalizations are enabled with:

```
  preprocessor normalize_tcp: ips
```

This will ensure consistency in retransmitted data. Any segments overlapping with previously received segments will have the overlaps overwritten to contain the data first received.

Readback:
0. From Sensor/ run ./readback.sh tcp_i?s.conf ../Source/tcp.pcap.

Playback:
1. On the sensor, run ./inline.sh tcp_i?s.conf.
2. On the sink, run ../recv.sh.
3. On the source, run ./send.sh tcp.pcap.
4. Type Ctl-C on the sensor and sink to terminate.

The results for `tcp_ids.conf` are (we are looking at packets 4 and 5, which have payload):

```
16:24:05.877457 10.1.2.3.48620 > 10.9.8.7.8: . [tcp sum ok] 1:11(10) ack 1 win 256 (ttl
64, id 4, len 50)
0x0000   4500 0032 0004 0000 4006 5caf 0a01 0203        E..2....@.\.....
0x0010   0a09 0807 bdec 0008 0000 0002 0000 0002        ................
0x0020   5010 0100 ed1f 0000 7365 676d 656e 743d        P.......segment=
0x0030   3120                                           1.

16:24:05.877458 10.1.2.3.48620 > 10.9.8.7.8: . [tcp sum ok] 6:16(10) ack 1 win 256 (ttl
64, id 5, len 50)
0x0000   4500 0032 0005 0000 4006 5cae 0a01 0203        E..2....@.\.....
0x0010   0a09 0807 bdec 0008 0000 0007 0000 0002        ................
0x0020   5010 0100 26fc 0000 5858 5858 5873 6567        P...&...XXXXXseg
0x0030   3d32                                           =2
```

The results for tcp_ips.conf are:

```
16:14:12.402351 10.1.2.3.48620 > 10.9.8.7.8: . [tcp sum ok] 1:11(10) ack 1 win 256 (ttl
64, id 4, len 50)
0x0000 4500 0032 0004 0000 4006 5caf 0a01 0203 E..2....@.\.....
0x0010 0a09 0807 bdec 0008 0000 0002 0000 0002 ................
0x0020 5010 0100 ed1f 0000 7365 676d 656e 743d P.......segment=
0x0030 3120 1.

16:14:12.402352 10.1.2.3.48620 > 10.9.8.7.8: . [tcp sum ok] 6:16(10) ack 1 win 256 (ttl
64, id 5, len 50)
0x0000 4500 0032 0005 0000 4006 5cae 0a01 0203 E..2....@.\.....
```

```
0x0010 0a09 0807 bdec 0008 0000 0007 0000 0002 ................
0x0020 5010 0100 6407 0000 6e74 3d31 2073 6567 P...d...nt=1.seg
0x0030 3d32 =2
```

In this session, the client sends this 10 octet payload “segment=1” starting at relative sequence number 1, and then sends this 10 octet payload “XXXXXseg=2” starting at relative sequence number 6. Without stream normalization, payload at sequences 6 through 10 would be delivered to the server twice, first as “nt=1 ” and then as “XXXXX”. The actual data delivered to the application would then depend on the server's TCP reassembly policy, which is implementation specific.

With stream normalization enabled, the payload at sequences 6 through 10 is the same in both packets. In other words, the server receives this 10 octet payload “segment=1 “ starting at relative sequence number 1 and then receives this 10 octet payload “nt=1 seg=2” starting at relative sequence number 6. Regardless of implementation, there is only one way to reassemble this stream.

```
Verdicts:
      Allow:            7 ( 43.750%)
      Block:            0 (  0.000%)
    Replace:            1 (  6.250%)
```

The counts show that there was one packet changed in the ips case.

Closing

There are several other normalizations available, especially for TCP. For more information, see README.normalize or the Snort manual. Stay tuned for additional posts covering additional features of Snort 2.9.0.