



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Network Monitoring and Threat Detection In-Depth (Security 503)"
at <http://www.giac.org/registration/gcia>

Snort 3.0 Beta 3 for Analysts

GIAC (GCIA) Gold Certification

Author: Doug Burks, doug.burks@gmail.com

Advisor: Joel Esler

Accepted: April 17th 2009

Abstract

Snort is a free and open source Intrusion Detection Systems (IDS). The next generation of Snort, version 3.0, is currently available in beta form. This paper will demonstrate how analysts can begin experimenting with Snort 3.0 today by manually compiling the source code or by simply downloading a preconfigured bootable CD. This paper will also discuss the design of Snort 3.0 and its new features, such as multithreading, native inline bridging, dynamic reconfiguration, and native IPv6 support.

Introduction

In order to understand the future of Snort, one must first understand its past. The original version of Snort was written by Marty Roesch in 1998. The current production version as of this writing (April 2009) is 2.8.4. Over the course of a decade and all of the releases in between, the evolution of Snort can be seen as part of the escalating arms race with those who would cause harm to our networks. As intruders find new methods of IDS evasion, Snort evolves to resist those methods. As attackers find new ways into our networks, Snort is pushed into service in increasingly complex configurations which may or may not have been envisioned by Marty Roesch back in 1998. As networks get bigger and faster, so do the attacker's botnets and the amount of attack traffic that they can send. As a consequence, the Snort developers must constantly come up with new ways of increasing the speed and efficiency at which Snort is able to analyze packets. Snort's ability to keep up with the rapidly changing threat landscape has resulted in it becoming quite popular.

As Snort has grown more and more popular and the number of installations worldwide has grown, so has the number of complex configurations and the increased demands on the core feature set. Though Snort has so far been successful in evolving to fulfill the needs that are placed upon it by the IDS community, there have been some long-standing limitations which could prevent Snort from being successful in the future. The first limitation of the Snort 2.x series is that it is single-threaded and, therefore, unable to take advantage of multiple cores/processors. Furthermore, if one part of the Snort process has to wait, then Snort will drop packets. An example of this is logging to a database. If Snort is configured to log directly to a database, then it has to wait for the database write to complete before it can go back to its real job of sniffing packets. This is currently avoided by configuring Snort to output to unified/unified2 format and then using a separate utility like barnyard to process the unified output and perform the database inserts. On a multiprocessor box, the Snort thread can operate on one processor with the barnyard thread operating on another, effectively imitating a multithreaded application.

The second limitation of Snort 2.x is that it was not originally designed to run inline. A separate project called Snort Inline was created for this capability. The inline

Doug Burks, doug.burks@gmail.com

functionality was eventually merged into the mainline Snort in Snort 2.3.0 RC1.

However, this was still limited in that it relied on IPTables instead of libpcap and thus would only work on operating systems that support IPTables (Sturges, 2009).

Additionally, Snort 2.x cannot be reconfigured dynamically. Configuring Snort is accomplished via the `snort.conf` configuration file. Any changes to the `snort.conf` file require a restart of the Snort process to take effect. This is especially problematic when running inline considering that a restart would cause packet loss.

These limitations are the impetus for Snort 3.0, which is currently in development and available in beta form. Snort 3.0 Beta 3 was released on April 1, 2009. Snort 3.0 is a framework which can take advantage of today's multi-core processors, run inline, and be reconfigured on the fly without requiring a restart. To understand how Snort 3.0 implements these new features, let's examine its design.

1. Snort 3.0 Architecture

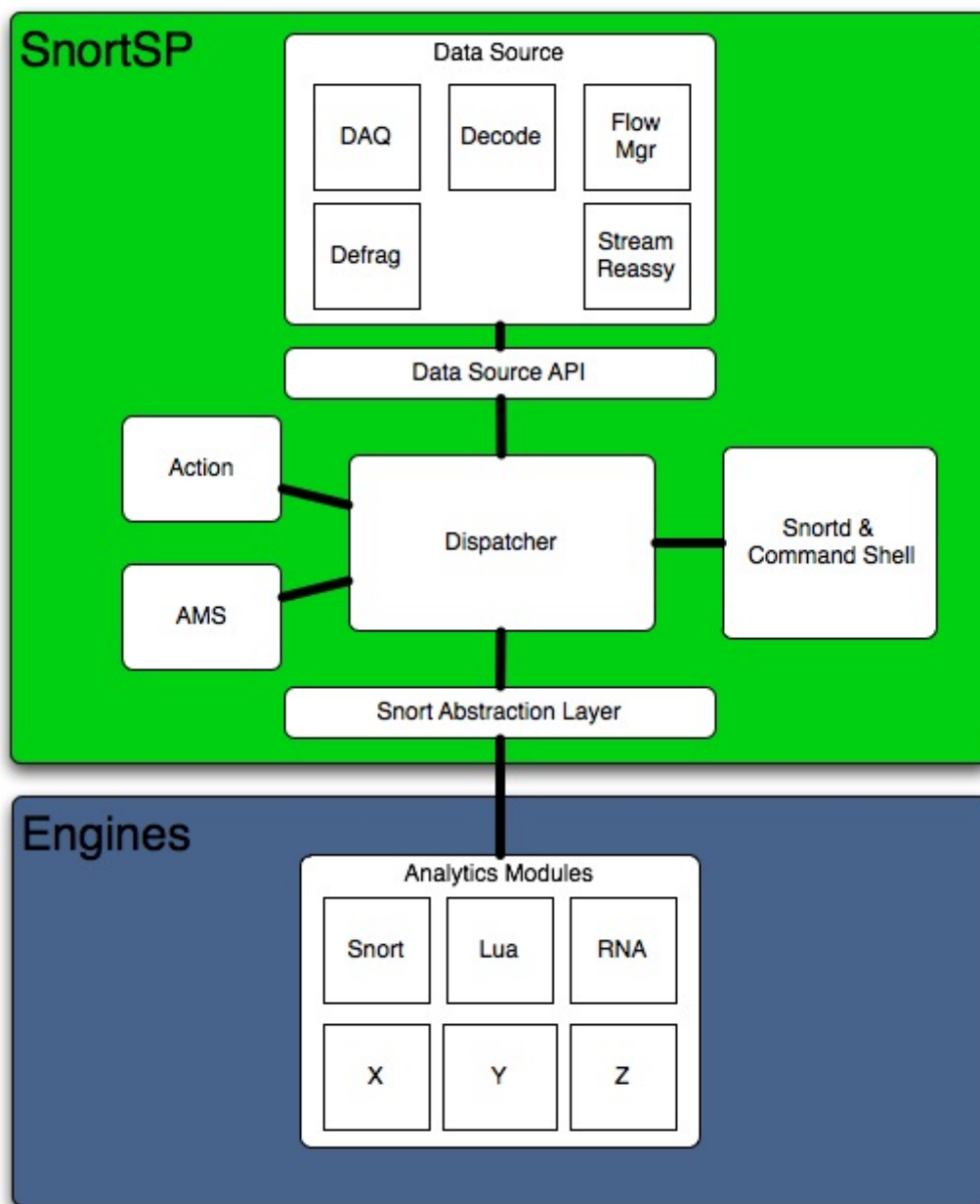


Figure 1-1: Snort 3.0 Architecture Diagram (Roesch, 2008c)

Figure 1-1 is Marty Roesch's diagram of the Snort 3.0 Architecture. Note that the Snort 3.0 Architecture consists of the Snort Security Platform (SnortSP) and engines that connect to that platform via the Snort Abstraction Layer. The Snort 3.0 Beta 3 tarball downloaded from the Snort website contains both SnortSP and a detection engine based

on Snort 2.8.3.1. The Snort 2.8.3.1 detection engine is currently the only analytics module available for SnortSP and is being used to help bridge the gap from Snort 2.x to Snort 3.0. It allows users to implement their existing Snort configuration on top of SnortSP. Expect to see more analytics modules when Snort 3.0 is released in final form. Since Snort 3.0 is multithreaded by default, each of the modules can simultaneously process the same traffic. This allows Snort 3.0 to take advantage of multiple cores/processors for increased speed and efficiency. It should be noted that the SnortSP Development Team performed exhaustive performance testing between Beta 2 and Beta 3 and consequently tweaked the multithreaded model (Roesch, 2009). Expect to see more changes and better performance by the final release of Snort 3.0.

Next, notice the module labeled "Snortd & Command Shell". Snort 3.0 includes a command shell based on the Lua scripting engine. This command shell can be used for dynamic reconfiguration. In terms of IDS tuning, Snort 2.x required a restart after any changes to the snort.conf file. This is still true in the current Snort 3.0 Beta when using the Snort 2.8.3.1 Detection Engine (the only detection engine currently available). Once Snort 3.0 native detection engines are available, then the analyst will have the ability to reconfigure them dynamically via the Lua console or by using snortsp_tool to connect to the socket interface.

Further enhancing the dynamic capabilities of SnortSP are pluggable Data Acquisition (DAQ) modules (seen in the Data Source module of the Architecture diagram). There are three DAQ modules currently available: file, pcap, and afpacket. This paper will demonstrate the use of the file DAQ module for reading pcap files, the pcap DAQ module for a traditional single Ethernet interface configuration, and the afpacket DAQ module to enable the new inline bridging functionality.

Now that we have a basic understanding of the design of Snort 3.0, let's begin experimenting with it to see how that design has been implemented.

2. Snort 3.0 LiveCD

Snort 3.0 has been integrated into a custom bootable CD that allows analysts to very easily experiment with Snort 3.0 and many other packet/security tools. The CD is

Doug Burks, doug.burks@gmail.com

called the Security Onion LiveCD and it should be available at the Snort website in the near future. Once available, an analyst can simply download the ISO image and boot it in a virtual machine, or burn it to a CD and reboot the computer with the CD in the drive.



Figure 2-1: Security Onion LiveCD Boot Menu

Figure 2-1 shows the boot menu. Pressing Enter at the boot menu will start the operating system and then load the graphical environment.

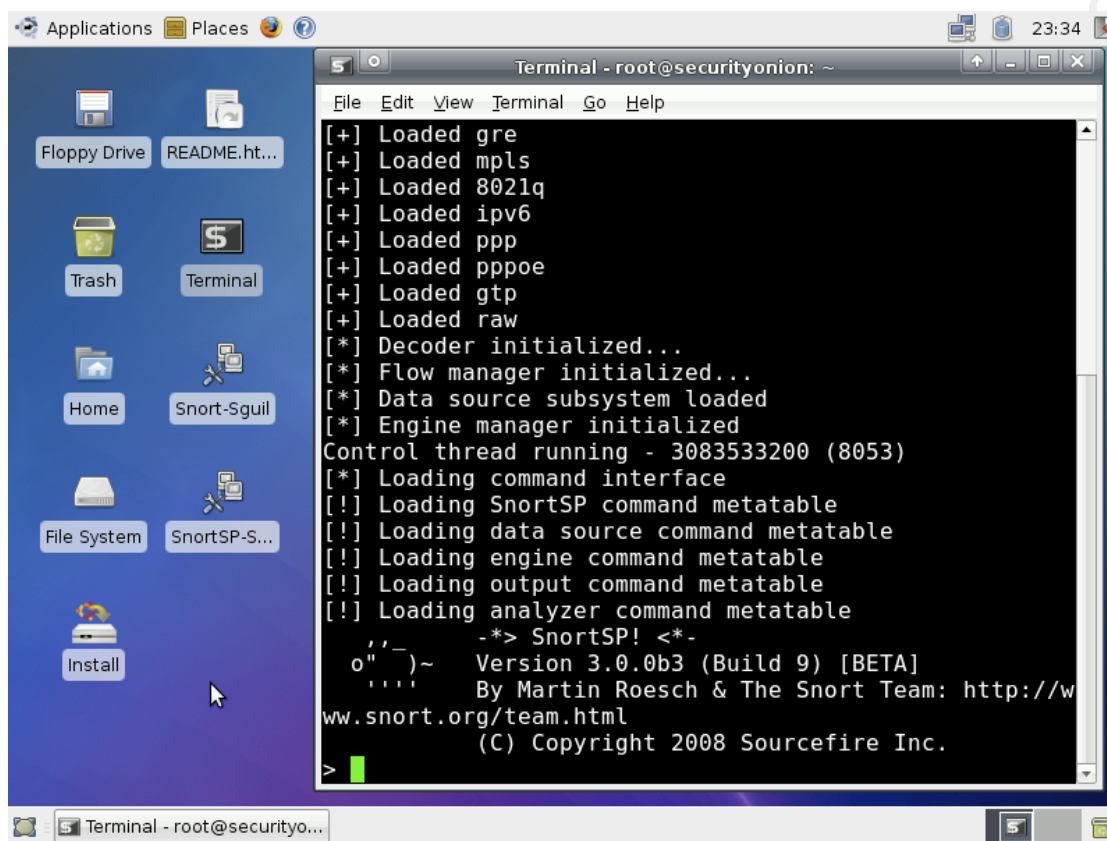


Figure 2-2: Security Onion LiveCD Desktop

Since Snort 3.0 has already been compiled and installed, an analyst can simply execute `snortsp` from the terminal (as shown in Figure 2-2) or by double-clicking the SnortSP-Sguil desktop shortcut. The SnortSP-Sguil desktop shortcut executes SnortSP and the Snort 2.8.3.1 detection engine, outputting alerts into Sguil as described later in this paper.

Also notice the desktop icon labeled "Install". This icon will launch the Ubuntu Ubiquity Installer which can be used to install the Security Onion LiveCD to a hard drive for a quick IDS installation. Full installation instructions can be found in the README file on the desktop.

For more information about the Security Onion LiveCD, please see Appendix A. Those planning on using the Security Onion distribution instead of manually compiling Snort 3.0 should still read through the following section. Doing so will aid in the

understanding of not only how Snort 3.0 was compiled and installed in the Security Onion distribution, but also in how the components of Snort 3.0 itself fit together.

3. Compiling and Installing Snort 3.0

Before Snort 3.0 can be compiled and installed, the analyst must ensure that all dependencies are met. The dependencies for Snort 3.0 are as follows: Lua 5.1.1 or better, Libdnet 1.10 or higher, a recent Libpcap, and a UUID library (Roesch, 2008b). Many modern Linux distributions will have these available in their repositories for easy installation. Installing these prerequisites on Ubuntu 8.04 is as easy as:

```
sudo -i

aptitude update

aptitude -y install build-essential \
libdumbnet1 libdumbnet-dev \
uuid uuid-dev \
libncurses5 libncurses5-dev \
libreadline5 libreadline5-dev \
libpcap0.8 libpcap0.8-dev \
libpcre3 libpcre3-dev \
liblua5.1-0 liblua5.1-0-dev \
flex bison
```

Once all dependencies have been satisfied, Snort 3.0 can be compiled. As mentioned in the previous section, the Snort 3.0 Beta 3 tarball contains the SnortSP framework and the Snort 2.8.3.1 detection engine. Note that an analyst could install the SnortSP framework by itself and skip the Snort 2.8.3.1 detection engine if they weren't planning on using Snort 2.x rules and alerts. However, since Snort 2.8.3.1 is the only detection engine currently available for SnortSP, installing just the SnortSP framework

would result in a sniffer with no alerting capability. Therefore, most analysts will want to compile and install both the SnortSP framework and the Snort 2.8.3.1 detection engine.

First, we'll download the tarball and unpack it:

```
cd /usr/local/src/

wget http://www.snort.org/dl/prerelease/3.0.0b3/
snortsp-3.0.0b3.tar.gz

tar zxvf snortsp-3.0.0b3.tar.gz

cd snortsp-3.0.0b3/
```

We're almost ready to begin compiling, but some systems may experience libtool problems during the compilation process. This can occur if /bin/sh is symlinked to /bin/dash instead of /bin/bash. We can fix this with a quick one-liner:

```
rm /bin/sh && ln -s /bin/bash /bin/sh
```

Now that we've satisfied libtool, we'll install SnortSP using the traditional `./configure; make; make install`. SnortSP is multi-threaded by default so just running `./configure` will configure for multi-threaded mode. As mentioned in the RELEASE.NOTES file, SnortSP can be configured for single-threaded mode with the `--enable-single-threaded` option.

```
./configure

make

make install
```

SnortSP has some new configuration files that we'll copy to `/etc/snortsp/`:

```
mkdir /etc/snortsp/

cp etc/* /etc/snortsp/
```

Next, we move on to the Snort 2.8.3.1 detection engine:

```
cd src/analysis/snort/

./configure \
```

```
--with-platform-includes=/usr/local/include \
```

```
--with-platform-libraries=/usr/local/lib
```

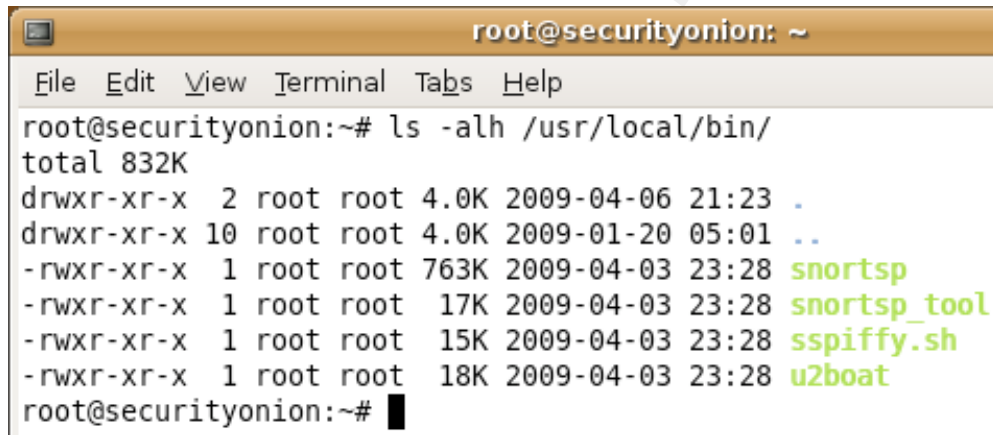
```
make
```

```
make install
```

Finally, Ubuntu installations need to update their shared libraries:

```
ldconfig
```

Now that installation is complete, let's examine the binaries that were installed in /usr/local/bin/:



```
root@securityonion: ~
File Edit View Terminal Tabs Help
root@securityonion:~# ls -alh /usr/local/bin/
total 832K
drwxr-xr-x  2 root root 4.0K 2009-04-06 21:23 .
drwxr-xr-x 10 root root 4.0K 2009-01-20 05:01 ..
-rwxr-xr-x  1 root root 763K 2009-04-03 23:28 snortsp
-rwxr-xr-x  1 root root  17K 2009-04-03 23:28 snortsp_tool
-rwxr-xr-x  1 root root  15K 2009-04-03 23:28 sspiffy.sh
-rwxr-xr-x  1 root root  18K 2009-04-03 23:28 u2boat
root@securityonion:~#
```

Figure 3-1: Snort 3.0 Binaries

Of course, snortsp is the core executable that we're interested in. Executing it with no options will start the SnortSP framework and the Lua command shell, but running it with the "-C" or "-D" options just starts the SnortSP framework with no command shell. In these modes, snortsp_tool can be used to connect to the socket interface and send configuration commands to the running SnortSP. The next tool is sspiffy.sh, a script used to convert existing Snort 2.x snort.conf files to be used with SnortSP and the Snort 2.8.3.1 detection engine. The sspiffy.sh script outputs two files: a new snort.conf file for the Snort 2.8.3.1 detection engine and a snort.lua file that will configure SnortSP to instantiate the Snort 2.8.3.1 detection engine using that newly created snort.conf file. The final executable is u2boat, a tool used to convert the new unified2 output to a standard pcap file format.

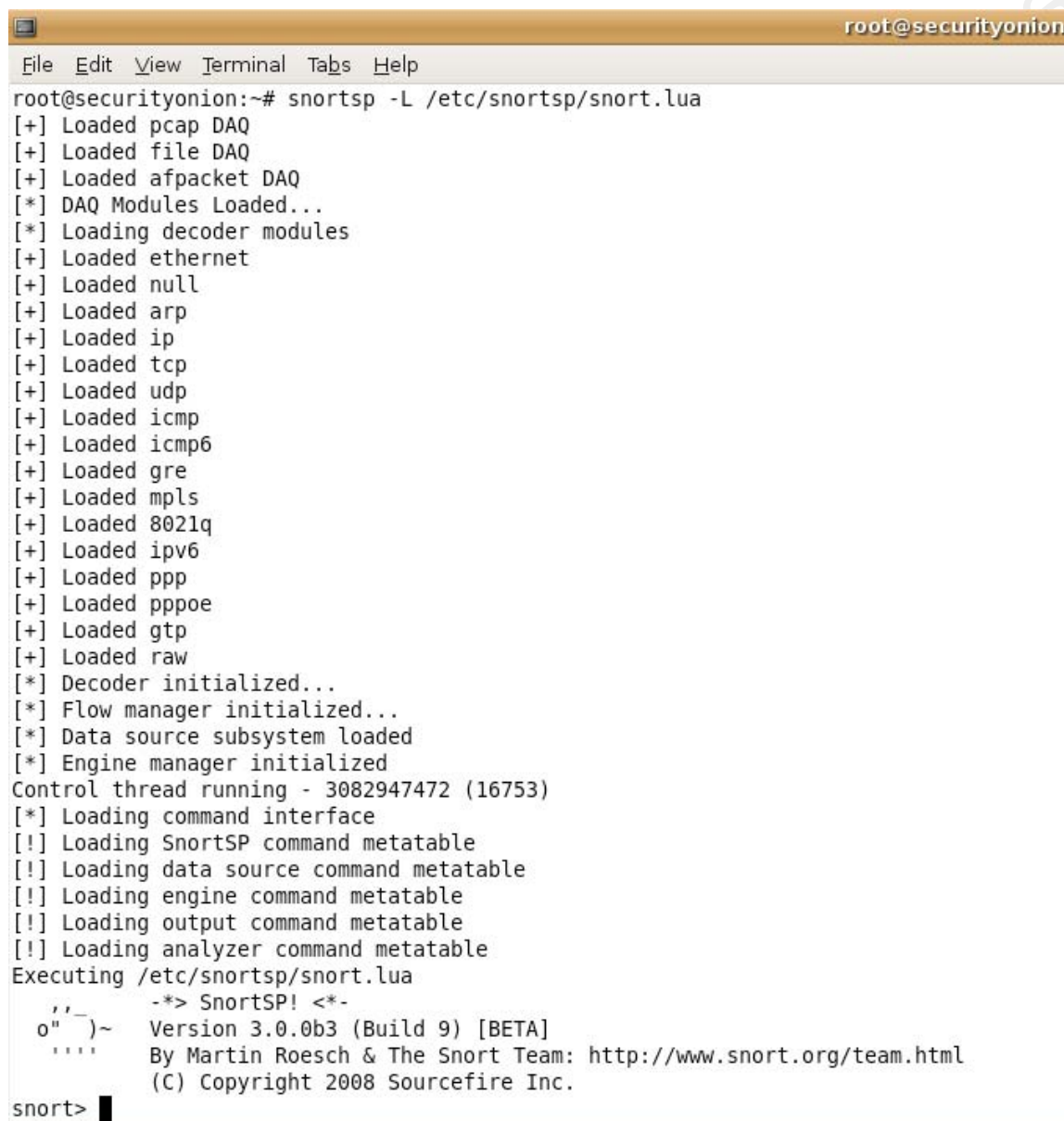
In this section, we covered how to compile and install SnortSP. Now that SnortSP is installed, let's see it in action!

4. SnortSP Data Acquisition (DAQ) Modules

To see SnortSP in action, we need to get some packets into it. As discussed in the Architecture section, this is done using data acquisition (DAQ) modules. We're going to look at all three of the currently available DAQ modules: file, pcap, and afpacket. To do this, we first need to start snortsp and have it load the Lua configuration file that was copied to /etc/snortsp/ during installation:

```
snortsp -L /etc/snortsp/snort.lua
```

SnortSP will load its modules, execute the snort.lua file, and then start the Lua command shell, as shown in Figure 4-1.



```

root@securityunion:~# snortsp -L /etc/snortsp/snort.lua
[+] Loaded pcap DAQ
[+] Loaded file DAQ
[+] Loaded afpacket DAQ
[*] DAQ Modules Loaded...
[*] Loading decoder modules
[+] Loaded ethernet
[+] Loaded null
[+] Loaded arp
[+] Loaded ip
[+] Loaded tcp
[+] Loaded udp
[+] Loaded icmp
[+] Loaded icmp6
[+] Loaded gre
[+] Loaded mpls
[+] Loaded 8021q
[+] Loaded ipv6
[+] Loaded ppp
[+] Loaded pppoe
[+] Loaded gtp
[+] Loaded raw
[*] Decoder initialized...
[*] Flow manager initialized...
[*] Data source subsystem loaded
[*] Engine manager initialized
Control thread running - 3082947472 (16753)
[*] Loading command interface
[!] Loading SnortSP command metatable
[!] Loading data source command metatable
[!] Loading engine command metatable
[!] Loading output command metatable
[!] Loading analyzer command metatable
Executing /etc/snortsp/snort.lua
    ,,_~
o"_)~  -*> SnortSP! <*-
'   '  Version 3.0.0b3 (Build 9) [BETA]
      By Martin Roesch & The Snort Team: http://www.snort.org/team.html
      (C) Copyright 2008 Sourcefire Inc.
snort>

```

Figure 4-1: SnortSP Lua Shell

Next, let's get a listing of the supported DAQ modules by typing the following in the Lua shell:

```
dsrc.list_daq_modules()
```

```

snort> dsrc.list_daq_modules()
The following is a list of the available DAQ modules:
  pcap
    Use eth<n> or bond<0> as the intf parameter or the filename parameter
    Packet based
    Capable of reading live interfaces
    Capable of reading a capture file
  file
    Use the filename parameter, the intf parameter is not used
    Packet based
    Capable of reading a capture file
  afpacket
    Use eth<n>[:eth<n>]... as the intf parameter
    Packet based
    Capable of reading live interfaces
    Capable of operating in inline mode
snort>

```

Figure 4-2: Output of `dsrc.list_daq_modules()`

In Figure 4-2, we can see the currently supported DAQ modules (pcap, file, and afpacket) and a description of each.

4.1. "file" DAQ module

Let's begin our tour of DAQ modules by using the "file" DAQ module. Please see Appendix E for a full listing of the default snort.lua file and notice the `runfile()` function. This function instantiates the "file" DAQ module and reads from a standard packet capture file. At the snortsp Lua shell, call the `runfile()` function and give it the name of the pcap file to be processed:

```
runfile("ping.pcap")
```

```

root@security
File Edit View Terminal Tabs Help
TCP Max flows: 262144
TCP Max idle: 30
TCP Memcap: 10000000
Other Max flows: 131072
Other Max idle: 30
Other Memcap: 1000000
[*] DAQ config:
  Filename: ping.pcap
  Snaplen: 1514
  Datalink: 1
  Count: 0
  Packet Count: 0
  File ptr: 0x80faf90
  analysis context ptr: 0x80e96c0
[*] Spawning engine thread!
e3 thread running - 3070897040 (16540)
[*] Packet 1 from file ping.pcap
[*] Packet Info
  Serial: 1
  Packet Time: 04/06-14:35:07.071698
  Packet Bytes: 98
  Captured Bytes: 96
  Layers: 2
  Flags: 00001000
[*] Ethernet (14 bytes)
  Source MAC Address: 08:00:27:26:70:81
  Dest MAC Address: 52:54:00:12:35:02
  Encapsulated Protocol: IPv4
[*] Payload (82 bytes)
0x0000: 45 00 00 54 00 00 40 00 01 20 8D 0A 00 02 0F E..T..@.@. ....
0x0010: 07 07 07 07 08 00 9A 46 94 40 00 01 1B 13 DA 49 .....F.@.....I
0x0020: E8 17 01 00 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 .....
0x0030: 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 ..... !"#
0x0040: 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F 30 31 32 33 $%&'()*+,-./0123
0x0050: 34 35 45

=====
e3 thread exiting - 3070897040 (16540)
snort>

```

Figure 4-3: Example Output of `runfile("ping.pcap")`

In Figure 4-3, we see SnortSP instantiating the "file" DAQ module and reading the file `ping.pcap` which contains a single ICMP Echo Request.

4.2. "pcap" DAQ module

Next, let's look at the "pcap" DAQ module. `snort.lua` contains a function called `sniff()` which will instantiate a "pcap" DAQ module on the interface that is specified as a parameter. Type the following to start sniffing packets (replacing "lo" with the interface to capture traffic from):

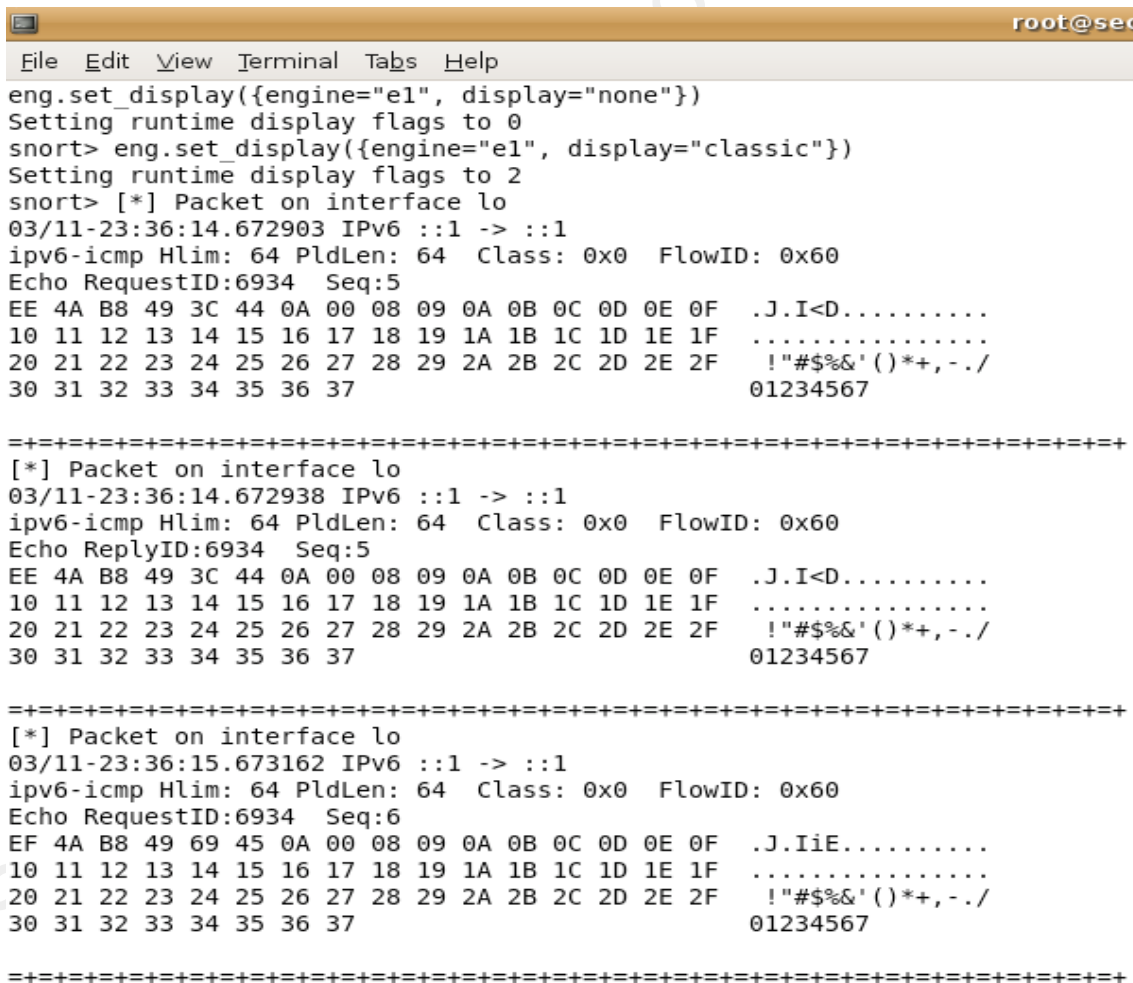
```
sniff("lo")
```

SnortSP is now capturing packets but not displaying them since that would interfere with the Lua shell. To see the packets, perform the following steps:

- Type the following:

```
eng.set_display({engine="e1", display="none"})
```

- Press the up arrow to retrieve the last command and change "none" to "classic".
- Watch the traffic.
- When finished, press the up arrow twice to retrieve the "none" command and press Enter.



```

root@sec
File Edit View Terminal Tabs Help
eng.set_display({engine="e1", display="none"})
Setting runtime display flags to 0
snort> eng.set_display({engine="e1", display="classic"})
Setting runtime display flags to 2
snort> [*] Packet on interface lo
03/11-23:36:14.672903 IPv6 ::1 -> ::1
ipv6-icmp Hlim: 64 PldLen: 64 Class: 0x0 FlowID: 0x60
Echo RequestID:6934 Seq:5
EE 4A B8 49 3C 44 0A 00 08 09 0A 0B 0C 0D 0E 0F .J.I<D.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#%&'()*+,-./
30 31 32 33 34 35 36 37 01234567

=====
[*] Packet on interface lo
03/11-23:36:14.672938 IPv6 ::1 -> ::1
ipv6-icmp Hlim: 64 PldLen: 64 Class: 0x0 FlowID: 0x60
Echo ReplyID:6934 Seq:5
EE 4A B8 49 3C 44 0A 00 08 09 0A 0B 0C 0D 0E 0F .J.I<D.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#%&'()*+,-./
30 31 32 33 34 35 36 37 01234567

=====
[*] Packet on interface lo
03/11-23:36:15.673162 IPv6 ::1 -> ::1
ipv6-icmp Hlim: 64 PldLen: 64 Class: 0x0 FlowID: 0x60
Echo RequestID:6934 Seq:6
EF 4A B8 49 69 45 0A 00 08 09 0A 0B 0C 0D 0E 0F .J.IiE.....
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F .....
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F !"#%&'()*+,-./
30 31 32 33 34 35 36 37 01234567

=====

```

Figure 4-4: SnortSP displaying IPv6 packets in classic mode

Figure 4-4 shows SnortSP capturing packets on lo (the loopback interface). These packets were created by running the following command:

```
ping6 ip6-localhost
```

This command will send ICMPv6 Echo Request packets to the loopback interface over IPv6. This highlights another important feature of Snort 3.0: native support for IPv6.

We've examined the two DAQ modules that are exposed in the default snort.lua file, so let's terminate the existing SnortSP session by typing the following:

```
snp.shutdown()
```

4.3. "afpacket" DAQ module

We're going to complete our tour of the three currently available DAQ modules by implementing the afpacket DAQ module to enable inline bridging mode. Let's briefly discuss why an analyst would want to run in inline bridging mode.

Traditionally, an IDS such as Snort receives traffic via network tap or a span port on a switch, as illustrated in Figure 4-5.

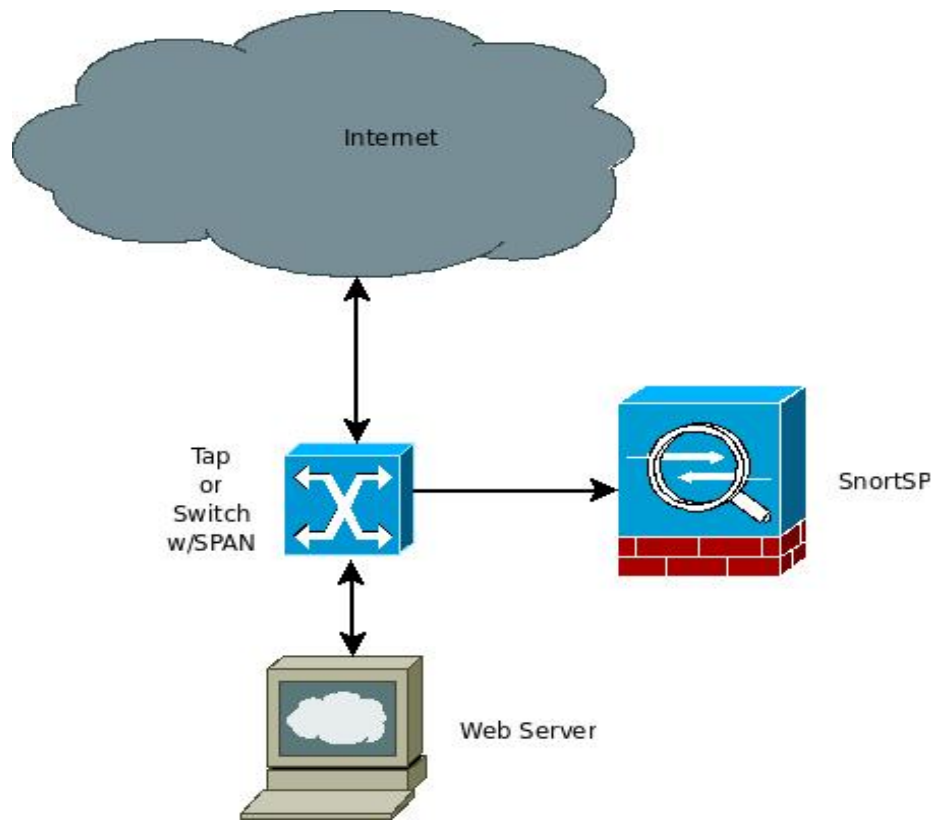


Figure 4-5: SnortSP in Traditional IDS Architecture Example

In Figure 4-5, all traffic from the Internet to the web server is copied to SnortSP. If an attacker sends an exploit to the web server and the Snort 2.8.3.1 detection engine has a rule that matches the attack, then SnortSP will alert. The IDS analyst can then see that an attack took place. If the IDS were actually in front of the web server (inline), and the attack matched one of the attack rules, then it could drop the traffic before it reached the web server. Thus, the Intrusion Detection System becomes an Intrusion Prevention System (IPS), as depicted in Figure 4-6.

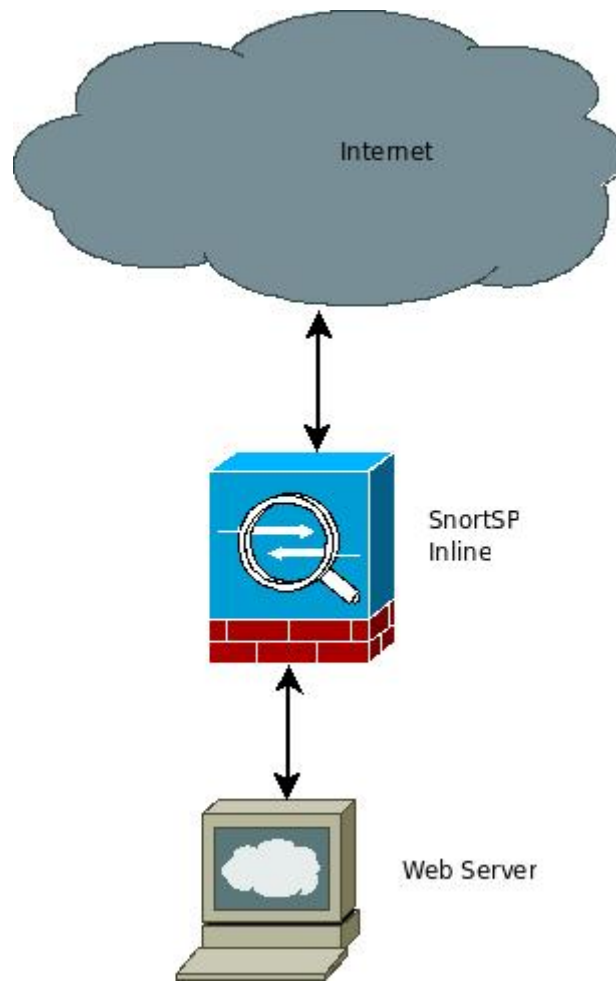


Figure 4-6: SnortSP Inline Example

As mentioned in the History section, the Snort 2.x series can run inline, but it requires IPTables. Snort 3.0, on the other hand, has native support for running inline via the `afpacket` DAQ module. Let's create a new file in the `/etc/snortsp/` directory called `bridge.lua` and add the code as seen in Appendix F. Let's examine the `bridge()` function itself:

```
function bridge (interface1, interface2)
if interface2 == nil then
error("Two interface strings must be specified")
end
dsrc1 = {name="src1",
```

```

type="afpacket",

intf=interface1..".."interface2,

flags=10,

snaplen=1514,

display="none",

tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},

other={maxflows=131072, maxidle=30, flow_memcap=10000000},

cksum_mode=0x0,

}

dsrc.new(dsrc1)

eng.new({name="e1"})

eng.link({engine="e1", source="src1"})

eng.start("e1")

end

```

The `bridge()` function is based on the `sniff()` function that we used earlier, so they are very similar. There are some differences, however. The first difference is the function definition itself—the `bridge()` function takes two parameters (two interfaces) instead of just one. The second difference is in the data source declaration (`dsrc1`). Here, `dsrc1` instantiates an `afpacket` DAQ module that creates an inline bridge (`flags=10`) from the two ethernet interfaces passed as parameters to the function. The `afpacket` `intf` variable is of the form `"eth0:eth1"` (Roesch, 2008b). Therefore, we construct our `intf` variable using the first interface parameter (`interface1`), the Lua concatenation operator (`".."`), a colon (`":"`), another Lua concatenation operator (`".."`), and the second interface parameter (`interface2`).

Now that we understand `bridge.lua`, start SnortSP and have it read the file using the following command:

```
snortsp -L /etc/snortsp/bridge.lua
```

SnortSP will start up, execute `bridge.lua`, and then start the SnortSP Lua shell. Type the following, replacing "eth0" and "eth1" with the interfaces to be bridged:

```
bridge("eth0", "eth1")
```

Once the bridge is up and running, SnortSP can be configured to display packets traversing the bridge:

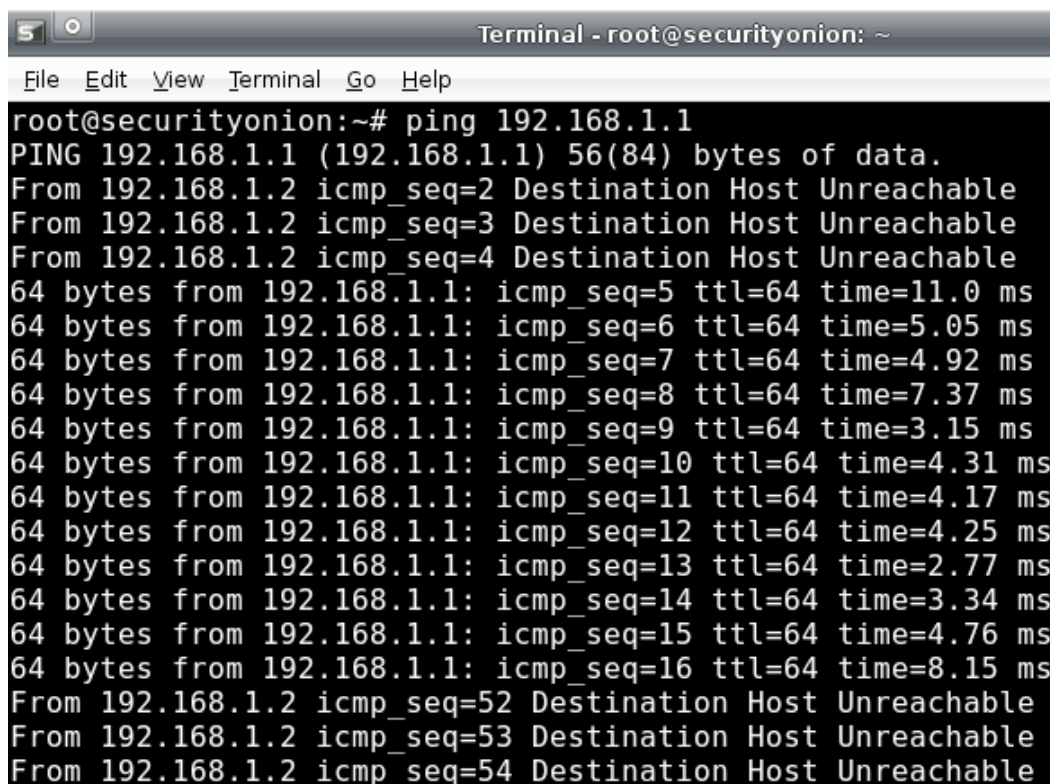
- Type the following:

```
eng.set_display({engine="e1", display="none"})
```
- Press the up arrow to retrieve the last command and change "none" to "classic".
- Watch the traffic.
- To stop viewing the traffic, press the up arrow twice to retrieve the "none" command and press Enter.

When finished with SnortSP, shut it down with the following command:

```
ssp.shutdown()
```

To test the inline bridging functionality, consider a SnortSP machine and two separate machines. The SnortSP machine has a management interface (eth0) and two interfaces with no IP addresses (eth1 and eth2). One of the test machines (test1) is configured with IP address 192.168.1.1 and is connected to eth1 of the SnortSP machine. The other test machine (test2) is configured with IP address 192.168.1.2 and is connected to eth2 of the SnortSP machine. On test2, ICMP Echo Requests were sent to test1 until three ICMP Destination Unreachable messages were received. The `bridge("eth1", "eth2")` function was then executed and 12 pings were allowed to cross the bridge. Finally, the bridge was taken down with a `"ssp.shutdown()"`, and ICMP Destination Unreachable messages then began appearing (see Figure 4-7).



```

Terminal - root@securityonion: ~
File Edit View Terminal Go Help
root@securityonion:~# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
From 192.168.1.2 icmp_seq=2 Destination Host Unreachable
From 192.168.1.2 icmp_seq=3 Destination Host Unreachable
From 192.168.1.2 icmp_seq=4 Destination Host Unreachable
64 bytes from 192.168.1.1: icmp_seq=5 ttl=64 time=11.0 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=64 time=5.05 ms
64 bytes from 192.168.1.1: icmp_seq=7 ttl=64 time=4.92 ms
64 bytes from 192.168.1.1: icmp_seq=8 ttl=64 time=7.37 ms
64 bytes from 192.168.1.1: icmp_seq=9 ttl=64 time=3.15 ms
64 bytes from 192.168.1.1: icmp_seq=10 ttl=64 time=4.31 ms
64 bytes from 192.168.1.1: icmp_seq=11 ttl=64 time=4.17 ms
64 bytes from 192.168.1.1: icmp_seq=12 ttl=64 time=4.25 ms
64 bytes from 192.168.1.1: icmp_seq=13 ttl=64 time=2.77 ms
64 bytes from 192.168.1.1: icmp_seq=14 ttl=64 time=3.34 ms
64 bytes from 192.168.1.1: icmp_seq=15 ttl=64 time=4.76 ms
64 bytes from 192.168.1.1: icmp_seq=16 ttl=64 time=8.15 ms
From 192.168.1.2 icmp_seq=52 Destination Host Unreachable
From 192.168.1.2 icmp_seq=53 Destination Host Unreachable
From 192.168.1.2 icmp_seq=54 Destination Host Unreachable

```

Figure 4-7: ICMP Echo Requests and Replies across SnortSP Bridge

Also, the SnortSP statistics in Figure 4-8 report that Snort 3.0 counted 24 ICMP packets (12 ICMP Echo Requests and 12 ICMP Echo Replies):

```

[*] AFPPacket DAQ config:
  Interface: eth1:eth2
  Snaplen: 1514
  Datalink: 1
  Page Size: 4096
  Pages: 32768
  Frame Size: 1584
  Frames/Page: 2
  Frames: 65536
  Count: 0
  Packet Count: 28
  analysis context ptr: 0x80e96b0
e1 thread exiting - 2801695632 (5906)
[*] INACTIVE data source src1 received 29 packets on eth1:eth2
  Analyzed: 29 (100.000%)
  Dropped: 0 (0.000%)
  Idle Cycles: 3
[-] Ethernet Stats:
  Count: 28
[-] ARP Stats:
  Count: 4
[-] IPv4 Stats:
  Count: 24
[-] ICMP Stats:
  Count: 24
[-] Raw Stats:
  Count: 28
  Bytes: 1416

```

Figure 4-8: SnortSP Inline Bridge Statistics

In this section, we explored the three DAQ modules that are currently available in Snort 3.0. These DAQ modules were demonstrated by interacting with the command shell built into snortsp. Another way to interact with snortsp is to connect to its socket interface using snortsp_tool.

5. Controlling SnortSP using snortsp_tool

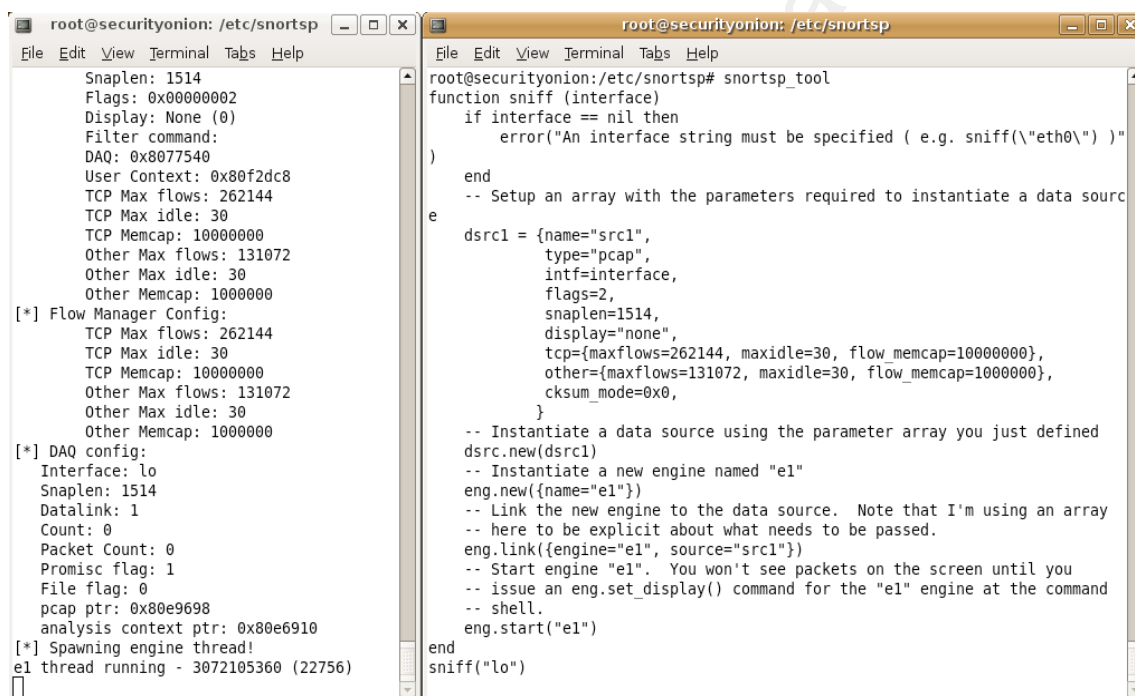
In a root terminal, start SnortSP by typing the following:

```
snortsp
```

In another root terminal, type the following to connect to the socket interface of snortsp:

snortsp_tool

Snortsp_tool will connect to the socket interface of the running snortsp process and allow the analyst to configure SnortSP. However, snortsp_tool isn't aware of the Lua file that was loaded at snortsp startup. So to use, for example, the sniff() function in the default snort.lua, the analyst will need to copy/paste that function into the snortsp_tool terminal before calling the sniff() function.



The image shows two terminal windows side-by-side. The left window, titled 'root@securityunion: /etc/snortsp', displays the configuration and status of the snortsp process. It shows various settings like Snaplen (1514), Flags (0x00000002), Display (None (0)), Filter command, DAQ (0x8077540), User Context (0x80f2dc8), TCP Max flows (262144), TCP Max idle (30), TCP Memcap (10000000), Other Max flows (131072), Other Max idle (30), Other Memcap (1000000), and Flow Manager Config. It also shows DAQ config (Interface: lo, Snaplen: 1514, Datalink: 1, Count: 0, Packet Count: 0, Promisc flag: 1, File flag: 0, pcap ptr: 0x80e9698, analysis context ptr: 0x80e6910) and spawning engine thread (e1 thread running - 3072105360 (22756)). The right window, titled 'root@securityunion: /etc/snortsp', shows the execution of the snortsp_tool script. The script defines a sniff function that takes an interface as an argument. It then instantiates a data source (dsrcl) with parameters like name, type, intf, flags, snaplen, display, tcp, and other. It then instantiates a new engine named 'e1', links it to the data source, starts it, and finally calls the sniff function with the argument 'lo'.

```

root@securityunion: /etc/snortsp
File Edit View Terminal Tabs Help
Snaplen: 1514
Flags: 0x00000002
Display: None (0)
Filter command:
DAQ: 0x8077540
User Context: 0x80f2dc8
TCP Max flows: 262144
TCP Max idle: 30
TCP Memcap: 10000000
Other Max flows: 131072
Other Max idle: 30
Other Memcap: 1000000
[*] Flow Manager Config:
TCP Max flows: 262144
TCP Max idle: 30
TCP Memcap: 10000000
Other Max flows: 131072
Other Max idle: 30
Other Memcap: 1000000
[*] DAQ config:
Interface: lo
Snaplen: 1514
Datalink: 1
Count: 0
Packet Count: 0
Promisc flag: 1
File flag: 0
pcap ptr: 0x80e9698
analysis context ptr: 0x80e6910
[*] Spawning engine thread!
e1 thread running - 3072105360 (22756)

root@securityunion: /etc/snortsp
File Edit View Terminal Tabs Help
root@securityunion:/etc/snortsp# snortsp_tool
function sniff (interface)
  if interface == nil then
    error("An interface string must be specified ( e.g. sniff(\"eth0\") )")
  end
  -- Setup an array with the parameters required to instantiate a data source
  dsrcl = {name="src1",
           type="pcap",
           intf=interface,
           flags=2,
           snaplen=1514,
           display="none",
           tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},
           other={maxflows=131072, maxidle=30, flow_memcap=1000000},
           cksum_mode=0x0,
           }
  -- Instantiate a data source using the parameter array you just defined
  dsrcl.new(dsrcl)
  -- Instantiate a new engine named "e1"
  eng.new({name="e1"})
  -- Link the new engine to the data source. Note that I'm using an array
  -- here to be explicit about what needs to be passed.
  eng.link({engine="e1", source="src1"})
  -- Start engine "e1". You won't see packets on the screen until you
  -- issue an eng.set_display() command for the "e1" engine at the command
  -- shell.
  eng.start("e1")
end
sniff("lo")

```

Figure 5-1: snortsp sniffing as directed by snortsp_tool

In Figure 5-1, snortsp is running in the left window and snortsp_tool is running in the right window. Snortsp_tool has connected to the running instance of snortsp, and the sniff() function has been copied from the default snort.lua file into the snortsp_tool window. The sniff("lo") function is then executed and snortsp begins sniffing on the loopback interface. When finished, snortsp_tool is used to send the ssp.shutdown() command to shut down SnortSP, after which snortsp_tool exits.

In this section, we demonstrated SnortSP receiving commands from snortsp_tool via socket. Ultimately, when running in production, snortsp would be started with the "-D" option to daemonize itself and then all runtime configuration would be performed

with `snortsp_tool`. If we're going to run in production, though, we don't just want to run a sniffer; we want to run in Intrusion Detection mode.

6. Snort 3.0 Intrusion Detection Mode

In Intrusion Detection mode, SnortSP captures packets and hands them off to the Snort 2.8.3.1 detection engine for analysis and alerting. The best open source tool to manage Snort alerts is Sguil and the easiest way to install Sguil is with NSMnow.

6.1. Installing Sguil using NSMnow

NSMnow automatically installs and configures barnyard2 (compatible with the Snort 3.0 unified2 output format), sancp, Sguil, and Snort 2.x. We're going to replace NSMnow's snort 2.x alert process with Snort 3.0. The first step is to download NSMnow and run it as follows:

```
mkdir /usr/local/src/NSMnow
cd /usr/local/src/NSMnow
wget http://www.securixlive.com/\
download/nsmnow/NSMnow-1.3.5.tar.gz
tar zxvf NSMnow-1.3.5.tar.gz
./NSMnow -i -y
```

Since Ubuntu uses AppArmor by default, AppArmor must be configured to allow MySQL to read the alert data that is produced by the Snort 2.8.3.1 detection engine:

```
if ! grep "/nsm/server_data/server1/load" \
/etc/apparmor.d/usr.sbin.mysql > /dev/null
then
sed -i 's|}| /nsm/server_data/server1/load/* r,|g' \
/etc/apparmor.d/usr.sbin.mysql
echo "}" >> /etc/apparmor.d/usr.sbin.mysql
```

fi

/etc/init.d/apparmor restart

Next, start all NSM services, but skip the Snort 2.x alert process (since we're going to use SnortSP's 2.8.3.1 detection engine in its place):

/usr/local/sbin/nsm --server --start

/usr/local/sbin/nsm_sensor_ps-start --skip-snort-alert

The output of these commands should look like Figure 6-1.

```

root@securityonion:~# /usr/local/sbin/nsm --server --start
Starting: server1
* starting: sguil server [ OK ]
root@securityonion:~# /usr/local/sbin/nsm_sensor_ps-start --skip-snort-alert
Starting: sensor1
* starting: pcap_agent (sguil) [ OK ]
* starting: sancp_agent (sguil) [ OK ]
* starting: snort_agent (sguil) [ OK ]
* starting: barnyard2 (spooler, unified2 format) [ OK ]
* starting: sancp (session data) [ OK ]
* starting: snort (full packet data) [ OK ]
* disk space currently at 37%
root@securityonion:~#

```

Figure 6-1: Output of Sguil services startup

6.2. Converting snort.conf using sspiffy.sh

Let's copy the NSMnow snort.conf file and rules files to a new directory called */etc/snortsp_alert/*:

mkdir /etc/snortsp_alert

cd /etc/snortsp_alert

cp -R /etc/nsm/sensor1/.*

mv snort.conf snort_orig.conf

We're ready to process the NSMnow snort.conf file with sspiffy.sh, creating a snort.lua file. However, there is a bug in this release of sspiffy.sh: when specifying an interface like eth0, sspiffy.sh configures the data source to use the "file" DAQ module. As we saw in the DAQ section, the "file" DAQ module can only read from packet capture

files, not live interfaces. To capture from eth0, we need to change the data source to use the "pcap" DAQ module. We can facilitate this situation in one of two ways:

-Use the broken `sspiffy.sh` and then fix the resulting `snort.lua` with a quick one-liner:

```
sspiffy.sh /usr/local -c snort_orig.conf -i eth0
```

```
sed -i 's/type="file"/type="pcap"/g' snort.lua
```

-OR-

-Fix the source of the problem by applying the `sspiffy.sh` patch found in Appendix G. (Thanks to Russ Combs of the SnortSP Development Team for confirming the bug and supplying this patch.) Save the contents of Appendix G into a file called `sspiffy.patch` and run the following commands:

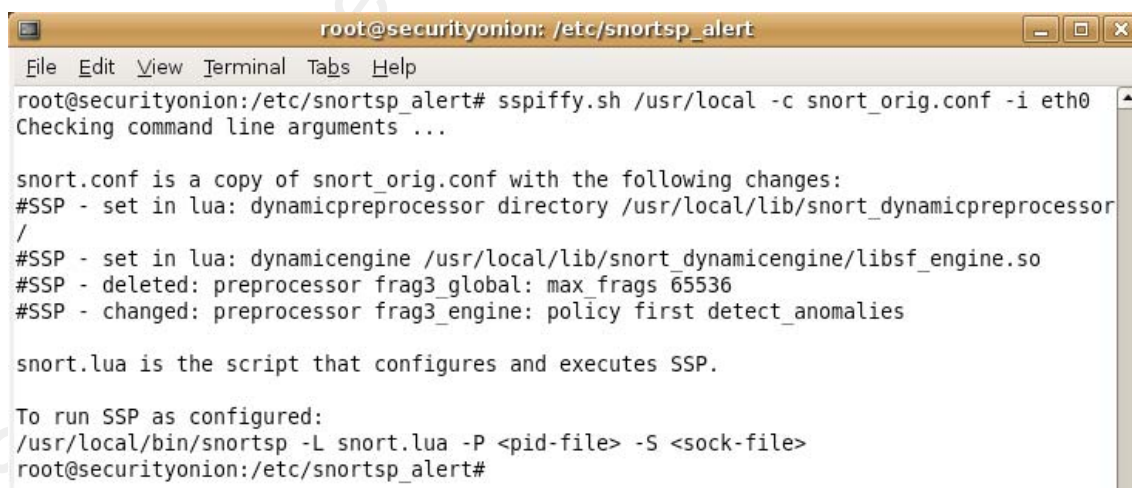
```
cd /usr/local/bin/
```

```
patch -p0 < sspiffy.patch
```

```
cd /etc/snortsp_alert
```

```
sspiffy.sh /usr/local -c snort_orig.conf -i eth0
```

In either case, the output of `sspiffy.sh` should look like Figure 6-2.



```

root@securityonion: /etc/snortsp_alert
File Edit View Terminal Tabs Help
root@securityonion:/etc/snortsp_alert# sspiffy.sh /usr/local -c snort_orig.conf -i eth0
Checking command line arguments ...

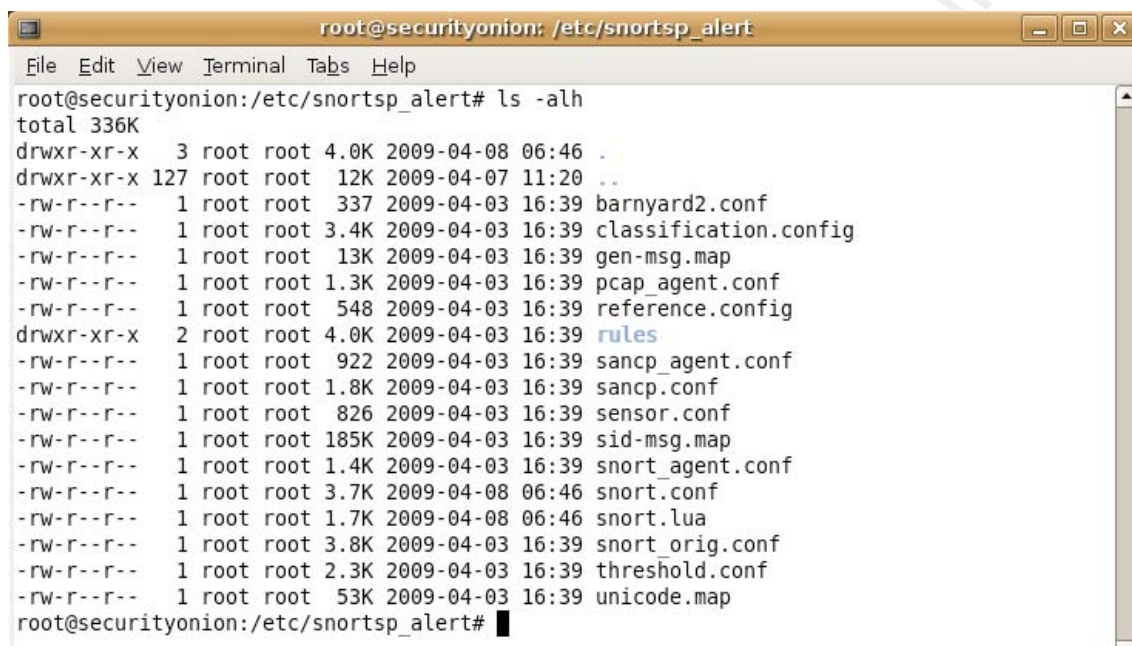
snort.conf is a copy of snort_orig.conf with the following changes:
#SSP - set in lua: dynamicpreprocessor directory /usr/local/lib/snort_dynamicpreprocessor
/
#SSP - set in lua: dynamicengine /usr/local/lib/snort_dynamicengine/libsf_engine.so
#SSP - deleted: preprocessor frag3_global: max frags 65536
#SSP - changed: preprocessor frag3_engine: policy first detect_anomalies

snort.lua is the script that configures and executes SSP.

To run SSP as configured:
/usr/local/bin/snortsp -L snort.lua -P <pid-file> -S <sock-file>
root@securityonion:/etc/snortsp_alert#
  
```

Figure 6-2: Output of `sspiffy.sh`

sspiffy.sh should have created two new files: snort.lua and a modified copy of snort_orig.conf called snort.conf. For reference, the final snort.lua and snort.conf can be seen in Appendices H and I, respectively. The /etc/snortsp_alert/ directory should look like Figure 6-3.



```

root@securityunion: /etc/snortsp_alert
File Edit View Terminal Tabs Help
root@securityunion:/etc/snortsp_alert# ls -alh
total 336K
drwxr-xr-x  3 root root 4.0K 2009-04-08 06:46 .
drwxr-xr-x 127 root root 12K 2009-04-07 11:20 ..
-rw-r--r--  1 root root  337 2009-04-03 16:39 barnyard2.conf
-rw-r--r--  1 root root  3.4K 2009-04-03 16:39 classification.config
-rw-r--r--  1 root root  13K 2009-04-03 16:39 gen-msg.map
-rw-r--r--  1 root root  1.3K 2009-04-03 16:39 pcap_agent.conf
-rw-r--r--  1 root root  548 2009-04-03 16:39 reference.config
drwxr-xr-x  2 root root 4.0K 2009-04-03 16:39 rules
-rw-r--r--  1 root root  922 2009-04-03 16:39 sancp_agent.conf
-rw-r--r--  1 root root  1.8K 2009-04-03 16:39 sancp.conf
-rw-r--r--  1 root root  826 2009-04-03 16:39 sensor.conf
-rw-r--r--  1 root root 185K 2009-04-03 16:39 sid-msg.map
-rw-r--r--  1 root root  1.4K 2009-04-03 16:39 snort_agent.conf
-rw-r--r--  1 root root  3.7K 2009-04-08 06:46 snort.conf
-rw-r--r--  1 root root  1.7K 2009-04-08 06:46 snort.lua
-rw-r--r--  1 root root  3.8K 2009-04-03 16:39 snort_orig.conf
-rw-r--r--  1 root root  2.3K 2009-04-03 16:39 threshold.conf
-rw-r--r--  1 root root  53K 2009-04-03 16:39 unicode.map
root@securityunion:/etc/snortsp_alert#

```

Figure 6-3: /etc/snortsp_alert/ directory listing

Next, we do some rule cleanup:

```

grep -v "sameip" rules/bad-traffic.rules > \
rules/bad-traffic.rules.2
rm -f rules/bad-traffic.rules
mv rules/bad-traffic.rules.2 rules/bad-traffic.rules

```

Finally, we start snortsp using the newly created snort.lua file. This snort.lua file configures snortsp to instantiate the Snort 2.8.3.1 detection engine using its new snort.conf file:

```
snortsp -C -L snort.lua
```

SnortSP is now capturing packets on eth0 and analyzing them with the Snort 2.8.3.1 detection engine. Let's verify that now. Launch the Sguil client by opening a new terminal and typing the following:

sguil.tk

When prompted, login to Sguil using the default credentials:

Username: sguil

Password: password

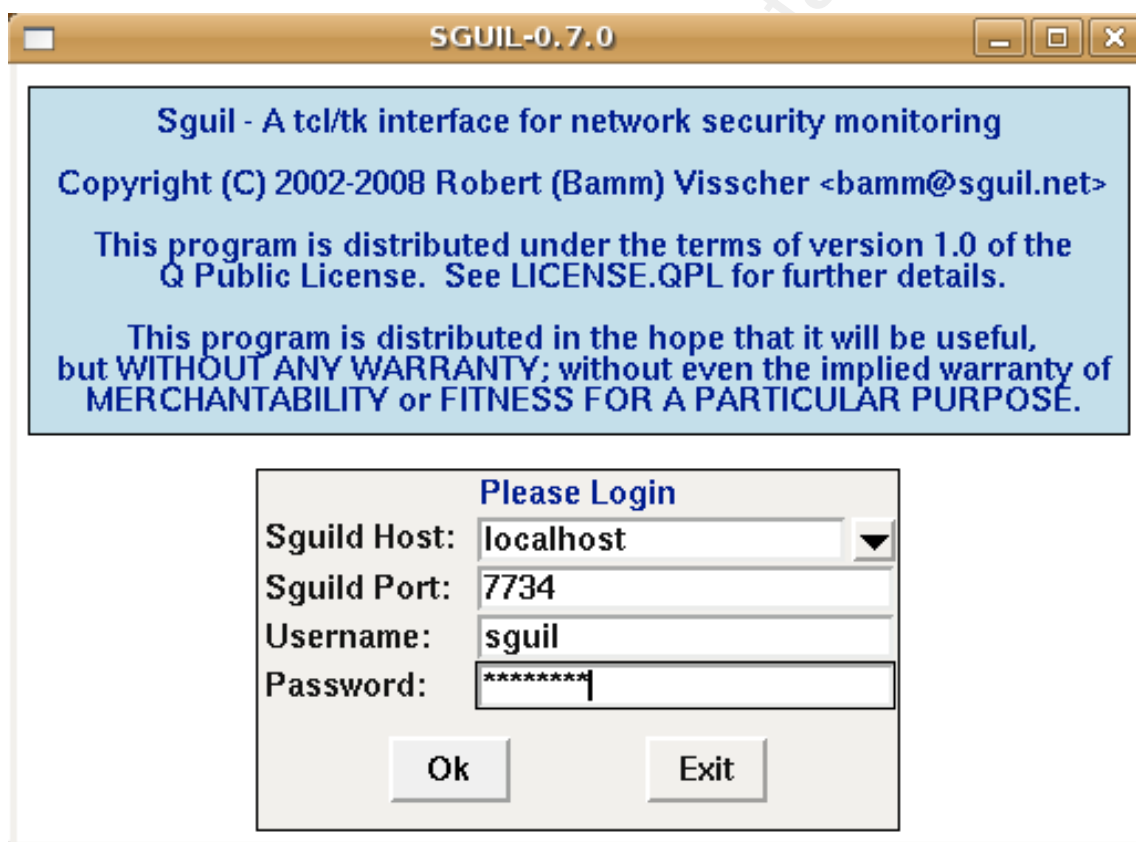


Figure 6-4: Sguil Login Window

Next, create some alerts by opening a browser and going to:

<http://www.testmyids.com>

The Sguil console should now display two new alerts with a source IP of 82.165.50.118 (the IP address of www.testmyids.com).

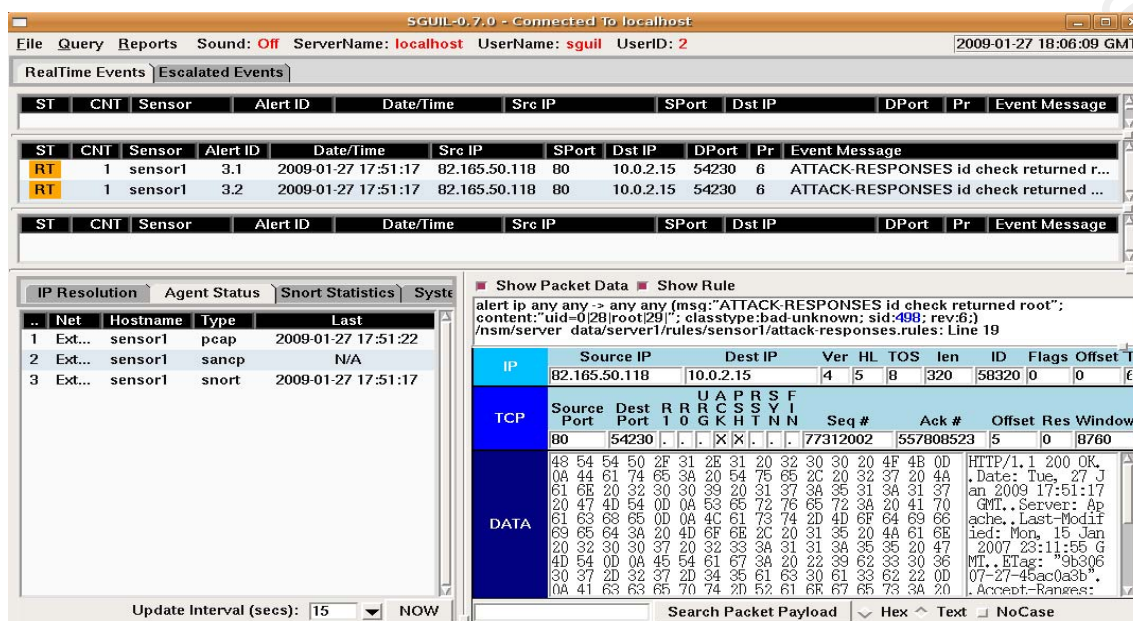


Figure 6-5: Sguil Console showing SnortSP/2.8.3.1 alerts

This demonstrates that SnortSP is capturing packets, analyzing them with the Snort 2.8.3.1 detection engine, and outputting in unified2 format, which is then read by Barnyard2 and inserted into the Sguil database.

When finished, close the Sguil console, return to the SnortSP window, and press Ctrl-c to terminate the SnortSP process. Then type the following to terminate all NSMnow processes:

```
/usr/local/sbin/nsm --all --stop
```

NSMnow uses Barnyard2 to process the unified2 output, but another method of extracting data from the unified2 output would be to use the u2boat utility to convert to the pcap format.

6.3. Converting unified2 output using u2boat

NSMnow configures Snort to output its alerts in unified2 format to the /nsm/sensor_data/sensor1/ directory. Let's go to that directory and process a unified2-formatted file using the u2boat utility included in Snort 3.0. Finally, we'll verify the resulting pcap using tcpdump:

```
cd /nsm/sensor_data/sensor1
```


u2boat snort.unified2. test.pcap*

tcpdump -nr test.pcap

```

root@securityonion: /nsm/sensor_data/sensor1
File Edit View Terminal Tabs Help
root@securityonion:/nsm/sensor_data/sensor1# ls -alh
total 24K
drwxrwxr-x 5 sguil sguil 4.0K 2009-04-07 06:08 .
drwxr-xr-x 3 root root 4.0K 2009-04-03 16:37 ..
drwxrwxr-x 4 sguil sguil 4.0K 2009-04-03 22:47 dailylogs
drwxrwxr-x 2 sguil sguil 4.0K 2009-04-03 16:37 portscans
drwxrwxr-x 2 sguil sguil 4.0K 2009-04-07 06:07 sancp
-rw-r--r-- 1 root root 0 2009-04-03 16:39 snort.stats
-rw-r--r-- 1 root root 918 2009-04-03 23:20 snort.unified2.1238815188
root@securityonion:/nsm/sensor_data/sensor1# u2boat snort.unified2.1238815188 test.pcap
Defaulting to pcap output.
root@securityonion:/nsm/sensor_data/sensor1# ls -alh
total 28K
drwxrwxr-x 5 sguil sguil 4.0K 2009-04-07 06:09 .
drwxr-xr-x 3 root root 4.0K 2009-04-03 16:37 ..
drwxrwxr-x 4 sguil sguil 4.0K 2009-04-03 22:47 dailylogs
drwxrwxr-x 2 sguil sguil 4.0K 2009-04-03 16:37 portscans
drwxrwxr-x 2 sguil sguil 4.0K 2009-04-07 06:07 sancp
-rw-r--r-- 1 root root 0 2009-04-03 16:39 snort.stats
-rw-r--r-- 1 root root 918 2009-04-03 23:20 snort.unified2.1238815188
-rw-r--r-- 1 root root 782 2009-04-07 06:09 test.pcap
root@securityonion:/nsm/sensor_data/sensor1# tcpdump -nr test.pcap
reading from file test.pcap, link-type EN10MB (Ethernet)
23:20:02.284294 IP 82.165.50.118.80 > 10.0.2.15.40133: P 209920002:209920311(309) ack 2212220492 win 8760
23:20:02.284294 IP 82.165.50.118.80 > 10.0.2.15.40133: P 0:309(309) ack 1 win 8760
root@securityonion:/nsm/sensor_data/sensor1#

```

Figure 6-6: u2boat converting unified2 output to pcap file

Figure 6-6 shows u2boat converting snort.unified2.1238815188 to a pcap file called test.pcap. tcpdump is then used to verify that test.pcap contains two records with source address 82.165.50.118 and destination address 10.0.2.15 (the same as the alerts that were displayed in the Sguil console).

In this section, we experienced Snort 3.0 in Intrusion Detection mode using the Snort 2.8.3.1 detection engine. This detection engine still requires configuration in snort.conf and reconfiguration still requires a restart of the detection engine. In the future, Lua will be used to build various traffic analysis applications, as evidenced by the commented-out lsniiff() function found in the default snort.lua file in Appendix E. Once these Lua-based detection engines are available, analysts will be able to use the Lua shell to perform all SnortSP configuration and tuning without requiring a restart.

7. Conclusion

In this paper, we have demonstrated the architectural changes in Snort 3.0 and the reason for these changes. We have also illustrated SnortSP's new features, such as native IPv6 support, multithreading, native inline bridging, and dynamic configuration using the Lua scripting language. However, these features are merely a glimpse of the future of Snort. In the near future, the Snort 3.0 development team will be hosting a public CVS server, with future beta releases introducing native Snort 3.0 detection engines and a new TCP stream management subsystem (Roesch, 2009). Analysts can, and should, contribute to the Snort community by thoroughly testing the beta releases and providing feedback to the Snort 3.0 development team.

8. References

- Roesch, M (2007). Snort 3.0 Architecture Series Part 1: Overview. Retrieved January 2, 2009, from Security Sauce Web Site:
<http://securitysauce.blogspot.com/2007/11/snort-30-architecture-series-part-1.html>
- Roesch, M (2008). SnortSP Introduction. Retrieved April 6, 2009, from Snort Web Site:
<http://www.snort.org/dl/snortsp/>
- Roesch, M (2008). SnortSP README. Retrieved April 6, 2009, from Snort Web Site:
<http://www.snort.org/dl/snortsp/README.txt>
- Roesch, M (2008). Snort 3.0 Architecture Series Part 2: Changes and Betas. Retrieved January 2, 2009, from Security Sauce Web site:
<http://securitysauce.blogspot.com/2008/08/snort-30-architecture-series-part-2.html>
- Roesch, M (2008). Snort 3.0 Architecture Series Part 3: The command shell. Retrieved January 2, 2009, from Security Sauce Web Site:
<http://securitysauce.blogspot.com/2008/08/snort-30-architecture-series-part-3.html>
- Roesch, M (2009). Snort 3.0 Beta 3 Released. Retrieved April 6, 2009, from Security Sauce Web Site: <http://securitysauce.blogspot.com/2009/04/snort-30-beta-3-released.html>

Doug Burks, doug.burks@gmail.com

Sturges, S (2009). Snort 2.8.3.2 Manual. Retrieved March 1, 2009, from Snort Web Site:

http://www.snort.org/docs/snort_htmanuals/htmanual_2832/node11.html

Appendix A: Building a Snort 3.0 LiveCD

The Security Onion LiveCD is based on Ubuntu 8.04 (plus all available updates) and includes Snort 3.0 Beta 3, Snort 2.8, Sguil, sanp, and many other packet/security tools. The ISO image was built using Reconstructor (<http://reconstructor.aperantis.com/>). Reconstructor makes it easy for analysts to build their own custom LiveCD. It uses modules to programmatically add and configure the software installed in the ISO image. (These modules are essentially just shell scripts with a few special variables.) The Reconstructor module used for installing/configuring Snort 3, NSMnow, and the Snort3/Sguil integration can be found in Appendices B, C, and D (respectively). For those who wish to build their own Snort 3.0 LiveCD, here's a brief overview:

- Download the Ubuntu 8.04 ISO image from:
<http://www.ubuntu.com/>
- Use the ISO image to install Ubuntu 8.04. This installation of Ubuntu will be the "build" machine for building the LiveCD. Do not delete the ISO image as we will need it later.
- Boot into the Ubuntu 8.04 installation. Download the latest version of Reconstructor (currently 2.8.1) from:
http://reconstructor.aperantis.com/index.php?option=com_remository&Itemid=33&func=select&id=5
- Decompress the Reconstructor tarball and then copy the three Reconstructor modules in the following Appendices to the modules directory.
- Launch Reconstructor with the following command:
sudo python reconstructor.py

- Follow the prompts and have Reconstructor import Ubuntu 8.04 from the ISO image.
- Use the Reconstructor root terminal to remove any unnecessary packages from the LiveCD and free up space.
- Have Reconstructor apply the three Snort 3 modules to the LiveCD in the following order. (Note that Internet access is required so that the modules can download packages from the Internet.)
 1. SnortSP
 2. NSMnow
 3. SnortSP/Sguil integration
- Perform any other LiveCD customization desired.
- Build the new ISO image.

For more detailed information on the Reconstructor build process, please see the comprehensive documentation on the Reconstructor website:

<http://reconstructor.wiki.sourceforge.net/>

Appendix B: mod-install-snortsp.rmod

```
#!/bin/sh
#
# Reconstructor Module - Install SnortSP
#   Copyright (c) 2006 Reconstructor Team <http://reconstructor.aperantis.com>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

RMOD_ENGINE=1.0
RMOD_CATEGORY='Software'
RMOD_SUBCATEGORY='Networking'
RMOD_NAME='SnortSP'
RMOD_AUTHOR='Doug Burks'
RMOD_VERSION=0.1
RMOD_DESCRIPTION='Downloads, compiles, and installs SnortSP'
RMOD_RUN_IN_CHROOT=True
RMOD_UPDATE_URL='http://reconstructor.aperantis.com/update/modules/'

# install snortsp
VER="3.0.0b3"
echo Running $RMOD_NAME...
rm /bin/sh && ln -s /bin/bash /bin/sh
aptitude update
aptitude -y install build-essential \
libdumbnet1 libdumbnet-dev \
uuid uuid-dev \
libncurses5 libncurses5-dev \
libreadline5 libreadline5-dev \
libpcap0.8 libpcap0.8-dev \
libpcre3 libpcre3-dev \
liblua5.1-0 liblua5.1-0-dev \
flex bison
```

Doug Burks, doug.burks@gmail.com

```
cd /usr/local/src/  
wget http://snort.org/dl/prerelease/$VER/snortsp-$VER.tar.gz  
tar zxvf snortsp-$VER.tar.gz  
rm -f snortsp-$VER.tar.gz  
cd snortsp-$VER  
./configure  
make  
make install  
mkdir -p /etc/snortsp/  
cp etc/* /etc/snortsp/  
cd src/analysis/snort  
./configure \  
--with-platform-includes=/usr/local/include \  
--with-platform-libraries=/usr/local/lib  
make  
make install  
ldconfig  
cd /usr/local/src/  
rm -rf snortsp-$VER  
  
# clean cache  
aptitude clean  
aptitude autoclean  
echo $RMOD_NAME Finished...  
exit 0
```

Appendix C: mod-install-NSMnow.rmod

```
#!/bin/sh
#
# Reconstructor Module - Install NSMnow
#   Copyright (c) 2006 Reconstructor Team <http://reconstructor.aperantis.com>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
```

```
RMOD_ENGINE=1.0
RMOD_CATEGORY='Software'
RMOD_SUBCATEGORY='Networking'
RMOD_NAME='NSMnow'
RMOD_AUTHOR='Doug Burks'
RMOD_VERSION=0.1
RMOD_DESCRIPTION='Installs NSMnow'
RMOD_RUN_IN_CHROOT=True
RMOD_UPDATE_URL='http://reconstructor.aperantis.com/update/modules/'
```

```
# Install the new version
VER="1.3.5"
FOLDER="NSMnow-$VER"
FILE="$FOLDER.tar.gz"
echo Running $RMOD_NAME...
aptitude update
aptitude -y install libclass-std-perl libconfig-std-perl \
libdigest-sha1-perl oinkmaster
mkdir -p /usr/local/src/$FOLDER/
cd /usr/local/src/$FOLDER/
wget http://www.securixlive.com/download/nsmnow/$FILE
tar zxvf $FILE
rm -f $FILE
./NSMnow -i -y
```

Doug Burks, doug.burks@gmail.com

```
# clean cache  
aptitude clean  
aptitude autoclean  
echo $RMOD_NAME Finished...  
exit 0
```

Appendix D: mod-install-snortsp-sguil.rmod

```
#!/bin/sh
#
# Reconstructor Module - Configure SnortSP/Sguil integration
#   Copyright (c) 2006 Reconstructor Team <http://reconstructor.aperantis.com>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License
# as published by the Free Software Foundation; either version 2
# of the License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

RMOD_ENGINE=1.0
RMOD_CATEGORY='Software'
RMOD_SUBCATEGORY='Networking'
RMOD_NAME='SnortSP/Sguil Integration'
RMOD_AUTHOR='Doug Burks'
RMOD_VERSION=0.1
RMOD_DESCRIPTION='Configures SnortSP/Sguil integration'
RMOD_RUN_IN_CHROOT=True
RMOD_UPDATE_URL='http://reconstructor.aperantis.com/update/modules/'

# Configure SnortSP/Sguil integration
echo Running $RMOD_NAME...
rm -rf /etc/snortsp_alert
mkdir /etc/snortsp_alert
cd /etc/snortsp_alert
cp -R /etc/nsm/sensor1/* .
mv snort.conf snort_orig.conf
sspiffy.sh /usr/local -c snort_orig.conf -i eth0
sed -i 's|type="file"|type="pcap"|g' snort.lua
grep -v "sameip" rules/bad-traffic.rules > rules/bad-traffic.rules.2
rm -f rules/bad-traffic.rules
mv rules/bad-traffic.rules.2 rules/bad-traffic.rules

echo $RMOD_NAME Finished...
```

Doug Burks, doug.burks@gmail.com

exit 0

© SANS Institute 2009, Author retains full rights.

Appendix E: Default snort.lua

```

require "eng"
require "dsrc"

_PROMPT='snort> '

-- This function will instantiate a data source and an engine, link
-- them and start sniffing. The only argument is the interface name
-- upon which to sniff specified as a string. For example:
--
-- snort> sniff("eth0")
function sniff (interface)
  if interface == nil then
    error("An interface string must be specified ( e.g. sniff(\"eth0\") )")
  end
  -- Setup an array with the parameters required to instantiate a data source
  dsrc1 = { name="src1",
            type="pcap",
            intf=interface,
            flags=2,
            snaplen=1514,
            display="none",
            tcp={ maxflows=262144, maxidle=30, flow_memcap=10000000 },
            other={ maxflows=131072, maxidle=30, flow_memcap=1000000 },
            cksum_mode=0x0,
          }
  -- Instantiate a data source using the parameter array you just defined
  dsrc.new(dsrc1)
  -- Instantiate a new engine named "e1"
  eng.new({ name="e1" })
  -- Link the new engine to the data source. Note that I'm using an array
  -- here to be explicit about what needs to be passed.
  eng.link({ engine="e1", source="src1" })
  -- Start engine "e1". You won't see packets on the screen until you
  -- issue an eng.set_display() command for the "e1" engine at the command
  -- shell.
  eng.start("e1")
end

-- This function will instantiate a data source and an engine, link
-- them and start sniffing. Arguments are the interface to sniff on
-- and a BPF filter to apply to the session (if any). To send a
-- "NULL" string as the BPF filter simply specify "" as the filter.
function fsniff (interface, bpf)
  if interface == nil then
    error("An interface string must be specified ( e.g. sniff(\"eth0\") )")
  end

```

```

end
dsrc2 = { name="src2",
          type="pcap",
          intf=interface,
          flags=2,
          command=bpf,
          snaplen=1514,
          display="max",
          tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},
          other={maxflows=131072, maxidle=30, flow_memcap=1000000},
          cksum_mode=0x0,
        }
dsrc.new(dsrc2)
eng.new({ name="e2" })
eng.link({ engine="e2", source="src2" })
eng.start("e2")
end

-- This function will instantiate a data source and an engine, link
-- them and start sniffing. The only argument is the interface name
-- upon which to sniff specified as a string. This function will also
-- load a Lua script file called snort-funcs.lua and call the function within
-- that file named "lua_analyzer" which just hexdumps the packet payload.
-- Use your imagination for applications of this lua-based traffic analysis
-- capability. Example:
--
-- snort> lsniiff("eth0")
--function lsniiff (interface)
--  if interface == nil then
--    error("An interface string must be specified ( e.g. sniff(\"eth0\") )")
--  end
--  dsrc3 = { name="src3",
--            type="pcap",
--            intf=interface,
--            flags=2,
--            snaplen=1514,
--            display="max",
--            tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},
--            other={maxflows=131072, maxidle=30, flow_memcap=1000000},
--            cksum_mode=0x0,
--          }
--  dsrc.new(dsrc3)
--  eng.new({ name="e3" })
--  eng.link({ engine="e3", source="src3" })
--  eng.lua_setup("e3", "/etc/snort_funcs.lua", "lua_analyzer")
--  eng.start("e3")

```

```

--end

function gtp_test(pcapfile)
  if pcapfile == nil then
    error("A filename string must be specified ( e.g. gtp_test(\"gtp_test.pcap\") )")
  end
  dsrc4 = { name="src4",
            type="file",
            intf="file",
            filename=pcapfile,
            flags=1,
            snaplen=1514,
            max_count=6,
            display="max",
            tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},
            other={maxflows=131072, maxidle=30, flow_memcap=10000000},
            cksum_mode=0x0,
          }
  gtp_support="enable",
  dsrc.new(dsrc4)
  eng.new({ name="e4" })
  eng.link({ engine="e4", source="src4" })
  eng.run_file("e4", pcapfile)
end

--function lua_engine_test(interface)
--  luaconf = { script = "etc/lua_eng.lua",
--              instance_name = "luaflow1",
--              type = flow }
--
--  lua_analytics1 = analytics.new("lua", "1.0", luaconf)
--  eng.new({ name="e3" })
--  eng.link_analytics("e4", lua_analytics)
--
--end

function runfile(pcapfile)
  if pcapfile == nil then
    error("A filename string must be specified ( e.g. gtp_test(\"gtp_test.pcap\") )")
  end
  dsrc3 = { name="src3",
            type="file",
            intf="file",
            filename=pcapfile,
            flags=1,
            snaplen=1514,

```

```
    display="max",
    tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},
    other={maxflows=131072, maxidle=30, flow_memcap=10000000},
    cksum_mode=0x0,
  }
  dsrc.new(dsrc3)
  eng.new({ name="e3" })
  eng.link({ engine="e3", source="src3" })
  eng.run_file("e3", pcapfile)
--  eng.unlink("e3")
--  eng.delete("e3")
--  dsrc.delete("src3")
end
```

Appendix F: bridge.lua

```
require "eng"  
require "dsrc"  
  
_PROMPT='snort> '  
  
function bridge (interface1, interface2)  
  if interface2 == nil then  
    error("Two interface strings must be specified")  
  end  
  dsrc1 = { name="src1",  
    type="afpacket",  
    intf=interface1.."":"..interface2,  
    flags=10,  
    snaplen=1514,  
    display="none",  
    tcp={maxflows=262144, maxidle=30, flow_memcap=10000000},  
    other={maxflows=131072, maxidle=30, flow_memcap=1000000},  
    cksum_mode=0x0,  
  }  
  dsrc.new(dsrc1)  
  eng.new({ name="e1"})  
  eng.link({ engine="e1", source="src1"})  
  eng.start("e1")  
end
```

Appendix G: sspiffy.patch

```
diff -u -B -b -r1.26 sspiffy.sh
--- sspiffy.sh 3 Dec 2008 22:54:33 -0000    1.26
+++ sspiffy.sh 6 Apr 2009 15:54:21 -0000
@@ -43,7 +43,7 @@
PREFIX=""
POLICY=""

-LOG_LEVEL="critical"
+LOG_LEVEL="info"

#-----
# capture command line arguments
@@ -619,10 +619,17 @@
    end
end
--
-function init_src (it, fn, fl)
+function init_src (ty, nm, fl)
+  if (ty=="file") then
+    it=ty
+    fn=nm
+  else
+    it=nm
+    fn=""
+  end
  dsrc.new({
    name=src,
-   type="file", snaplen=$SNAP,
+   type=ty, snaplen=$SNAP,
    intf=it, flags=fl,
    filename=fn, max_count=$MAXC,
    tcp={maxflows=$STREAM5_TCP, maxidle=$STREAM5_TCP_TO,
@@ -680,7 +687,7 @@
--
function run_live ()
  init(nan)
-  init_src("$INTF", "", $FLAG)
+  init_src("pcap", "$INTF", $FLAG)
  eng.start(egn)
end
--
```

Doug Burks, doug.burks@gmail.com

Appendix H: NIDS-mode snort.lua

```

egn="e1"
ani="an"
src="s1"
nan=1
--
snort="/usr/local/lib/snort/snort.so"
opttab={ dynamic_preprocessor_lib_dir="/usr/local/lib/snort/snort_preproc",
dynamic_engine_lib="/usr/local/lib/snort/sf_engine.so", conf="snort.conf" }
fragtab={ max_trackers=65536, policy="first" }
--
pcaps={ }

--
function init (num)
eng.new({ name=egn, cpu=0 })
for i=1,num do
--opttab["Z"] = "now" .. i
eng.add_analyzer({
engine=egn,
analyzer=ani .. i,
order=1,
module=snort,
data=opttab,
bpf="",
-- this cpu is ignored for single threaded builds
cpu=2,
--lb={ total=num, index=i-1 }
})
end
end
--
function init_src (it, fn, fl)
dsrc.new({
name=src,
type="pcap", snaplen=1514,
intf=it, flags=fl,
filename=fn, max_count=0,
tcp={ maxflows=8192, maxidle=30,
flow_memcap=10000000 },
other={ maxflows=131072, maxidle=30,
flow_memcap=1000000 },
defrag=fragtab,

```

Doug Burks, doug.burks@gmail.com

```

cksum_mode=0x3f,
display="none"
})
eng.link({engine=egn, source=src})
end
--
function term (num)
for i=1,num do
eng.rm_analyzer({
engine=egn,
analyzer=ani .. i
})
end
ssp.shutdown()
end
--
function term_src ()
eng.unlink(egn)
dsrc.delete(src)
end
--
function ana_cmd (op)
for i=1,nan do
ana=ani .. i
eng.cfg_analyzer({engine=egn, analyzer=ana, data={cmd=op}})
end
end
--
function run_test ()
analyzer.cfgtest({order=1, module=snort, data=opttab})
ssp.shutdown()
end
--
function run_file (pcap)
init_src("file", pcap, 2)
eng.test(egn)

term_src()
end
--
function run_files ()
init(nan)
for i,pcap in ipairs(pcaps) do
run_file(pcap)
end
term(nan)

```



```
end
--
function run_live ()
init(nan)
init_src("eth0", "", 2)
eng.start(egn)
end
--
function stats ()
ana_cmd("stats")
end
--
function reset ()
ana_cmd("reset")
end
--
function stop ()
eng.stop(egn)
term_src()
term(nan)
end
--
ssp.set_log_level("critical")
run_live()
```

Appendix I: NIDS-mode snort.conf

```
# snort.conf: auto-generated by NSMnow Administration on Mon Jan 26 12:34:53 EST
2009
var HOME_NET any
var EXTERNAL_NET any
var DNS_SERVERS $HOME_NET
var SMTP_SERVERS $HOME_NET
var HTTP_SERVERS $HOME_NET
var SQL_SERVERS $HOME_NET
var TELNET_SERVERS $HOME_NET
var SNMP_SERVERS $HOME_NET
portvar HTTP_PORTS 80
portvar SHELLCODE_PORTS !80
portvar ORACLE_PORTS 1521
var AIM_SERVERS
[64.12.24.0/23,64.12.28.0/23,64.12.161.0/24,64.12.163.0/24,64.12.200.0/24,205.188.3.0/
24,205.188.5.0/24,205.188.7.0/24,205.188.9.0/24,205.188.153.0/24,205.188.179.0/24,20
5.188.248.0/24]
var RULE_PATH ./rules
var PREPROC_PATH ./preproc_rules
config logdir: /nsm/sensor_data/sensor1
#SSP - set in lua: dynamicpreprocessor directory
/usr/local/lib/snort_dynamicpreprocessor/
#SSP - set in lua: dynamicengine /usr/local/lib/snort_dynamicengine/libsfe_engine.so
#SSP - deleted: preprocessor frag3_global: max_fragments 65536
preprocessor frag3: detect_anomalies
#SSP - changed: preprocessor frag3_engine: policy first detect_anomalies
preprocessor stream5_global: max_tcp 8192, track_tcp yes, track_udp no
preprocessor stream5_tcp: policy first, use_static_footprint_sizes
preprocessor perfmonitor: time 300 file /nsm/sensor_data/sensor1/snort.stats pktcnt
10000
preprocessor http_inspect: global iis_unicode_map unicode.map 1252
preprocessor http_inspect_server: server default profile all ports { 80 8080 8180 }
oversize_dir_length 500
preprocessor rpc_decode: 111 32771
preprocessor bo
preprocessor ftp_telnet: global encrypted_traffic yes inspection_type stateful
preprocessor ftp_telnet_protocol: telnet normalize ayt_attack_thresh 200
preprocessor ftp_telnet_protocol: ftp server default def_max_param_len 100
alt_max_param_len 200 { CWD } cmd_validity MODE < char ASBCZ > cmd_validity
MDTM < [ date nnnnnnnnnnnnnnn[n[n[n]]] ] string > chk_str_fmt { USER PASS RNFR
RNTD SITE MKD } telnet_cmds yes data_chan
```

Doug Burks, doug.burks@gmail.com

```

preprocessor ftp_telnet_protocol: ftp client default max_resp_len 256 bounce yes
telnet_cmds yes
preprocessor smtp: ports { 25 587 691 } inspection_type stateful normalize_cmds
normalize_cmds { EXPN VRFY RCPT } alt_max_command_line_len 260 { MAIL }
alt_max_command_line_len 300 { RCPT } alt_max_command_line_len 500 { HELP
HELO ETRN } alt_max_command_line_len 255 { EXPN VRFY }
preprocessor sfportscan: proto { all } memcap { 10000000 } sense_level { low }
preprocessor dcerpc: autodetect max_frag_size 3000 memcap 100000
preprocessor dns: ports { 53 } enable_rdata_overflow
preprocessor ssl: noinspect_encrypted
output unified2: filename snort.unified2, limit 128
include classification.config
include reference.config
include $RULE_PATH/local.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/exploit.rules
include $RULE_PATH/scan.rules
include $RULE_PATH/finger.rules
include $RULE_PATH/ftp.rules
include $RULE_PATH/telnet.rules
include $RULE_PATH/rpc.rules
include $RULE_PATH/rservices.rules
include $RULE_PATH/dos.rules
include $RULE_PATH/ddos.rules
include $RULE_PATH/dns.rules
include $RULE_PATH/tftp.rules
include $RULE_PATH/web-cgi.rules
include $RULE_PATH/web-coldfusion.rules
include $RULE_PATH/web-iis.rules
include $RULE_PATH/web-frontpage.rules
include $RULE_PATH/web-client.rules
include $RULE_PATH/web-php.rules
include $RULE_PATH/web-attacks.rules
include $RULE_PATH/sql.rules
include $RULE_PATH/x11.rules
include $RULE_PATH/icmp.rules
include $RULE_PATH/netbios.rules
include $RULE_PATH/misc.rules
include $RULE_PATH/attack-responses.rules
include $RULE_PATH/oracle.rules
include $RULE_PATH/mysql.rules
include $RULE_PATH/snmp.rules
include $RULE_PATH/smtp.rules
include $RULE_PATH/imap.rules
include $RULE_PATH/pop2.rules
include $RULE_PATH/pop3.rules

```

```
include $RULE_PATH/nntp.rules  
include $RULE_PATH/other-ids.rules  
include $RULE_PATH/icmp-info.rules  
include $RULE_PATH/experimental.rules
```