Building wireless IDS systems using open source.

cclark@quadrantsec.com

As a security researcher, penetration tester, and all around network security guy, Kismet has always had a special place in my heart when it comes to network security testing tools. When I'm on-site, doing an internal penetration test or network security audit, it is not uncommon to see Kismet running on my laptop. Sometimes it is simply out of general curiosity to see "what's out there", it might also be to determine if a "rogue access point" is operating, or I might be looking for a way to build a "covert channel" (ie - back door) out of the network. It's also possible that I'm just bored, and want to see what is flowing through the air.

I've been a Kismet user for a long time, and it is one of the many "tools" in my "tool box". If you're not familiar with Kismet, check out http://www.kismetwireless.net.

Kismet is basically a passive wireless network/device discovery tool. When Kismet is properly configured, you simply fire it up and it will tell you what wireless devices are in your area (802.11b/a/g/n/DECT).

You might recall that back in August 2010, Google got into a little bit of trouble using this utility. Basically, those goofy looking Google "streetview" cars that take pictures of streets were doing a little more than that. Using Kismet and GPS data, they were collecting information about wireless networks, along with packet payload data, which led them into trouble.

As a "security" guy, I tend to 'ride the fence' of exploitation versus protection. As much fun as it is to discover exploitable services on a network (legally!), it is also important to understand and realize how to "protect" the network.

Recently, while playing with Kismet, I began to ponder "Wireless Intrusion Detection Systems". I recalled that the Kismet web site said the following:

> "Kismet is an 802.11 layer2 wireless network detector, sniffer, and **intrusion detection system**."

Note the bold on "intrusion detection system". This led me to my latest exercise. That is, building a "Wireless IDS" (Intrusion Detection System) using "off the shelf" supplies. There's already a commercial market for wireless IDS systems. I'm not terribly interested in those, and I decided to explore the "Do It Yourself" avenue, so the software needed to be open source. Kismet was my first natural choice for this project.

All roads led back to Kismet

I did, however, decide to research other possible projects that might be more focused on wireless IDS. My criterium, of course, was that the project be "open source". To my surprise, there's really not much being done in the open source world that deals with wireless IDS. For example, I found the project widz.

"widz" looked interesting, but the problem is that this project hasn't been updated in quite some time (Feb. 2003). I also ran into "Snort Wireless" project references. This project appeared to be defunct. After some thought, I decided to stick with the well known, good old, Kismet. My thoughts where that I'd want something that has a code base that is actively being developed. Dragorn, the author of Kismet, has been doing this for quite some time and is still actively developing Kismet.

Goals

The primary idea for this "Proof of Concept" (PoC) system is that it has the potential of being used "out in the field". That is, in real world senarios.

I had several ideas about how I wanted to do this. The first involved using something like a WRT54G Linux based wireless access point. There were a couple of problems with this solution. First, I would need the system to act like a wireless "Access Point" (AP) and monitor wireless frequencies for layer 2 attacks. That means that the system would need a minimum of two internal radios. Second, I would also need enough horse power to do packet inspection among other things. My thoughts were that a small Linux based router would probably be under-powered for the task. I had also thought about a stand-alone machine that would "join" a network and passively monitor for layer 2 attacks. A little bit of research showed that this probably wouldn't work terribly well.

What I finally decided on doing was to take a PC, load Linux on it and turn it into a wireless AP. This way, I have the CPU power for packet analysis, with plenty of storage and RAM to load the software I might need. This offered a lot of flexibility, and it was ultimately the way I decided to go.

Another design concept which was critically important was for all data (alerts) to be dumped in a unified, single console for viewing. A system that uses several pieces of software that dumps data to various places wasn't an option. My PoC design had to have a single place where I could view all security related events. These events could be from Kismet, packet inspection software (Snort), or syslog (Sagan).

Hardware/tools

- One Linux based PC. I have a Pentium 4 3.00 ghz machine with a couple of gigs of RAM. This seemed ideal. At Quadrant Information Security we use Gentoo to build hardened systems. I'll be doing the same for this article, but distribution really shouldn't matter. The idea is to build a wireless IDS/IPS using the same methodology as our core Quadrant Information Security IDS/IPS system.

- Two EnGenuis 'Ultra Long range' wireless PCI adapters. These cards are based off the Atheros AR5001X+ chipset and are supported under Linux (madwifi-ng drivers and in kernel support). These are also capable of 600mW transmission power which is ideal since we'll be building an AP. I got these cards for about 20.00 bucks each.

(Figure 1. Still in the box Atheros "long range" wireless cards")

(Figure 2. Both cards out of the box)

- Beer - Since this article was written on the weekend, this was a nessesary requirement. It makes DIY projects, that don't involve high voltage/current electricity, more fun.

- Pauldotcom security weekly playing in the background. This is by far one of the more fun information security podcasts around. It's also good 'back ground sec related' noise :)

Loading the Atheros 5xxx drivers

The first, most logical step, was to make the Linux based PC into an AP. Keep in mind, I actually have two EnGenuis PCI wireless cards in the system. One will be used to build our AP, and the other for detection of layer 2 attacks (rogue access points, spoofed AP's, etc). Before we can get to that point, we need to load the appropriate drivers for these cards. On my soon-to-be wireless IDS system, a 'lspci'

showed:

```
01:01.0 Ethernet controller: Atheros Communications Inc. Atheros AR5001X+ Wireless Network Adapter (rev 01)
01:02.0 Ethernet controller: Atheros Communications Inc. Atheros AR5001X+ Wireless Network Adapter (rev 01)
```

The kernel I'll be using on my Linux based IDS/IPS system is the Gentoo 2.6.35-r2 hardened ebuild. We typically build our field IDS/IPS systems with harden kernels, so I decided it would be best to do the same with my PoC system. This means that pax/grsec is enabled and working. The idea is that this is a 'security' device, so any assistance in thwarting attacks can help.

The 2.6.35-r2 kernel has "Atheros 5xxx wireless card support", and this is what we'll be using. Below are the device/wireless kernel configurations I used.

```
Device Drivers  --->
 [*] Network device support  --->
   [*]   Wireless LAN  --->
     < M >   Atheros Wireless Cards  --->
       < M >   Atheros 5xxx wireless cards support
         [*]     Atheros 5xxx debugging
```

You'll probably want to install the wireless tools. This will give you commands like 'iwconfig', which may be needed for debugging. On my Gentoo based system, I simply typed:

```
# emerge wireless-tools
```

Check your distribution for the wireless tools. They'll likely be avaliable. Once your kernel has been compiled, you can load the Atheros 5k drivers by typing:

```
# modprobe ath5k
```

Check 'dmesg' for any errors. Make sure the module sees both wireless cards. Your 'dmesg' should look something like this, if completed successfully:

```
ath5k 0000:01:01.0: PCI->APIC IRQ transform: INT A -> IRQ 22
ath5k 0000:01:01.0: registered as 'phy0'
ath5k phy0: Atheros AR2414 chip found (MAC: 0x79, PHY: 0x45)
ath5k 0000:01:02.0: PCI->APIC IRQ transform: INT A -> IRQ 17
ath5k 0000:01:02.0: registered as 'phy1'
ath5k phy1: Atheros AR2414 chip found (MAC: 0x79, PHY: 0x45)
```

Buiding the AP

For the AP, we'll be using 'hostap'. The idea is to use one of our two wireless cards and switch it from a 'client' [Managed] mode, to 'AP' [Master] mode. This turns the computer into a wireless router. The card I'll be using is the first card (device: wlan0). Remember, we'll be using the second Atheros card for other purposes. I'm using hostapd version 6.9 for stability reasons. In the Gentoo world, to install hostapd, we would type:

```
# emerge hostapd
```

Check your distribution for information about possible hostapd packages.

Many people who use hostapd build a network 'bridge' between the wireless card and their internal network card. I will not be running hostapd in 'bridge' mode, but rather in NAT mode. This means I'll be assigning my AP card (wlan0) an IP address.

```
# ifconfig wlan0 10.220.0.1 netmask 255.255.255.0  # My home network
```

I'll be using this AP just as I would any other one. This means that WPA2 encryption is a must. In this case, I'll be using a pre-shared key. Your configuration may differ, so alter as you see fit. This is simply a 'base' configuration for you to start off with. In my case, the hostapd.conf file is stored in the /etc/hostapd/hostapd.conf.

```
interface=wlan0   # Our first Atheros card
driver=driver=nl80211  # Kernel level driver we're using
logger_syslog=-1  # This will be useful later!
logger_syslog_level=2  # ^^ for Sagan!
logger_stdout=-1
logger_stdout_level=2
dump_file=/tmp/hostapd.dump
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
ssid=secure   # Your network SSID (name)
hw_mode=g   # We'll restrict to wireless G mode only
channel=6   # On channel 6
beacon_int=100
dtim_period=2
max_num_sta=255
rts_threshold=2347
fragm_threshold=2346
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2    # Security settings (WPA encryption enabled)
wpa_passphrase=mysecretkey # Replace with your secret key!
wpa_key_mgmt=WPA-PSK
wpa_pairwise=CCMP
```

To start up hostapd.conf in 'debug' mode, simply type:

```
# hostapd -dd /etc/hostapd/hostapd.conf
```

Note: You might be wondering why we're not manually putting 'wlan0' in 'Master' mode with 'iwconfig'. You can do this, but 'hostapd' automatically does this at startup.

This will cause hostapd to run in the foreground, which can be useful for debugging. Without the '-dd' options, hostapd will become a system daemon (background). Since we're trying to build this system as a 'real access' point, you'll probably want to run DHCP services on the AP/wireless card (wlan0). If you're using the ISC (Internet Systems Consortium) DHCP server, you would configure the DHCP server just as you would any other. Here's a short example configuration:

```
# Set to your domain name.
option domain-name "quadrantsec.com";
# I typically run a caching DNS server on the local system.
option domain-name-servers 10.220.0.1;
default-lease-time 600;
max-lease-time 7200;
ddns-update-style ad-hoc;
log-facility local7;

subnet 10.220.0.0 netmask 255.255.255.0 {
  range 10.220.0.240 10.220.0.254;
  option routers 10.220.0.1;
}
```

Start your DHCP server and test. From another machine equipped with wirelss networking, you should see the SSID 'secure'. You should also be able to get a DHCP lease from your new DHCP server. Once the AP is deemed "operational", tweak your firewall settings and NAT to how you like them. That's outside of the scope of this document, so lets continue forward.

Back hall communications

Since we're attempting to build a Wireless IDS system we'll need to build a secure means to communicate with our back end server(s). This is ultimately where all our IDS/IPS alert information will end up and where we'll have a unified console to tie everything together. We *could* just send the data across the Internet to our back end unencrypted, but that would be a bad thing. Even though this is a PoC idea, we might as well do it right.

For the back end communications, at Quadrant Information Security. we utilize OpenVPN quite a bit. It's pretty simple to setup, and will give us an encrypted layer to transport our IDS events.

To build from source, simply go to http://openvpn.net and follow the instructions. If you're doing this on a Linux platform, odds are your distribution has a repository/build for OpenVPN. On a Gentoo system, you would simply type:

```
$ sudo emerge openvpn
```

For Ubuntu, you'd type:

```
$ sudo apt-get install openvpn
```

Configuration of OpenVPN is pretty straightforward, but I'll go over the basics of my PoC install. OpenVPN will be operating in a client/server model. That is, the clients will be our remote wireless IDS systems and they'll connect to our server, which is our event data storage/correlation back end. We'll basically be building a "point-to-point" VPN (encrypted) tunnel for our IDS data to flow to.

To start off, I'm going to create an OpenVPN configuration file on the "server" side.

```
#############################################################################
# "Server" side OpenVPN configuration.  Remember,  OpenVPN uses the tap/tun  #
# device.  This means you'll need to have tap/tun support in your kernel!    #
#############################################################################
#
dev tun
#
# Local side is 1.1.2.70,  remote will be 1.1.2.80
#
ifconfig 1.1.2.70 1.1.2.80
#
# The pre-shared "key" we'll use for encryption.
#
secret /etc/openvpn/wireless-ids.key
#
# The UDP port to listen on.  OpenVPN uses UDP by default,  we're going
# keep it that way.
#
port 2600
#
# Privilege separation.
#
user nobody
group nobody

# When the tunnel is considered "down"....
ping 15
ping-restart 45
ping-timer-rem
persist-tun
persist-key
verb 3
```

```
#
persist-tun
persist-key
verb 3
```

The client side is somewhat similar, with a few things switched around and added.

```
###############################################################################
# "Client" side of the connection.  This configuration file would be used on #
# the "remote" wireless IDS systems.                                          #
###############################################################################
#
dev tun
#
# Local side is 1.1.2.80,  remote will be 1.1.2.70
# Note:  It is reverse from the server side.
#
ifconfig 1.1.2.80 1.1.2.70
#
# The pre-shared "key" we'll use for encryption.
#
secret /etc/openvpn/wireless-ids.key
#
# The "target" or "server" we want to connect to.  This is where the
# "server" is "listening".
#
remote 12.145.241.1
#
# The remote "target" or "server" port we want to connect to (via
# UDP).
#
port 2600
#
# Privilege separation.
#
user nobody
group nobody
#
# When the tunnel is considered "down"....
ping 15
ping-restart 45
ping-timer-rem
persist-tun
persist-key
verb 3
```

Once the configuration files are in place, you'll need to generate a pre-shared key for OpenVPN to use. This key will go on both the client and the server. Keep in mind, it is a "secret" pre-shared key, so you'll want to transport the key over some sort of "secure" medium (For example, OpenSSH). To generate the key, type:

```
# openvpn --genkey --secret wireless-ids.key
```

Verify that both sides have a copy of the key, and test the connection. To test on both sides, type:

```
# openvpn --config /etc/openvpn/wireless-ids.conf
```

With any luck, you'll see something like the below:

```
Sun Sep 26 15:18:06 2010 Re-using pre-shared static key
Sun Sep 26 15:18:06 2010 Preserving previous TUN/TAP instance: tun1
Sun Sep 26 15:18:06 2010 Data Channel MTU parms [ L:1544 D:1450 EF:44 EB:4 ET:0 EL:0 ]
Sun Sep 26 15:18:06 2010 Local Options hash (VER=V4): '034192ac'
Sun Sep 26 15:18:06 2010 Expected Remote Options hash (VER=V4): 'a0e614ba'
Sun Sep 26 15:18:06 2010 Socket Buffers: R=[114688->131072] S=[114688->131072]
Sun Sep 26 15:18:06 2010 UDPv4 link local (bound): [undef]:2600
Sun Sep 26 15:18:06 2010 UDPv4 link remote: 12.145.241.1:2600
Sun Sep 26 15:18:16 2010 Peer Connection Initiated with 12.145.241.1:2600
Sun Sep 26 15:18:17 2010 Initialization Sequence Completed
```

To further verify, "ping" across the OpenVPN tunnel. The "server" side will be assigned the TCP/IP address 1.1.2.70. The client side will be assigned 1.1.2.80. If you can successfully "ping" across the OpenVPN connection, we have a secure means to transport our wireless IDS data. Also, after testing the link, you'll probably want to run OpenVPN in a "daemon" mode (background). To do this, simply append the "--daemon" flag. You might want to consider adding this to your startup in case either side gets rebooted. This way, the OpenVPN tunnel will automatically be established.

You might be curious about the IP structure we're using in the PoC wireless IDS system. Why didn't we use RFC1918 style addresses? We actually avoid RFC1918 addresses (for example, 192.168.0.0/24) because we don't know what type of networks these sensors might be installed in. It's likely that the remote wireless IDS system might be installed in a network already using RFC1918 addressing. This could potentially interfere with our wireless IDS system routing. With this in mind, we create a "point-to-point" tunnel using non-RFC1918 addresses to avoid any future conflicts. Remember, this is a point-to-point tunnel, so only two IP addresses are being used. Odds are in our favor that we won't be taking any "important" addresses.

Snort'ing the interface(s)

One part of our wireless IDS system will be used to monitor for attacks from within our network at the packet level. To do this, we'll be using a software from Sourcefire known as Snort. The very basic idea behind Snort is that it monitors traffic on a network interface for

security related events. For example, Snort might monitor the network for possible exploit attempts or people probing the network (nmap, etc) for information. Snort accomplishes this by using rule sets. When the rule sets match a 'bad' thing, Snort is able to send an 'alert' letting the administrators know that a possible attack is under way. At Quadrant we've used Snort for quite a while and have become fans of it. In many cases, we use Snort as our IDS/IPS engine. If you don't have experience with Snort and you're in the security field, I highly recommend checking it out.

For the purpose of this article, I'll be giving the basic overview of how we'll be using Snort for our wireless IDS system. Actually, I'm going to be configuring Snort to operate on more than just the wireless interface. My PoC wireless IDS system has the wireless network cards but is also connected to the Internet on a wired interface know as "eth1". So, for better protection of the network and the machines it controls, I'll be running Snort on the Internet facing network device (eth1) and on the wireless AP device (wlan0). On the Internet-facing side, we'll monitor for external attacks to our wireless IDS sensor. On the wireless LAN/AP side, we'll monitor for internally based attacks. The Internet facing "sensor" (Snort) will be known as "wireless-internet" (sensor id [sid]: # 1 on device eth1). The internal wireless LAN's "sensor" (Snort) will be known as "wireless-internal" (sensor id: # 2 on device wlan0).

While Snort will alert us when suspicious activity or attacks happen from within or externally from our wireless IDS system, we'll be running a firewall on both interfaces to hopefully slow or prevent the attack from being successful. The firewall configuration will largely depend on the type of installation you want to do. For example, it may be desired to egress firewall on the wireless/AP (wlan0) devices. Configuring firewalls with Linux has been a topic that's largely documented and outside of the scope of this document. I'll let you determine how you want to handle traffic on you wireless IDS system.

I typically build Snort directly from source. I find that many distributions typically ship older version of Snort, or distribute it with options I don't like. If you're not comfortable building Snort from source, check your distribution's repository. You might have better luck than me and be able to use a pre-built package of Snort. The Sourcefire team, the makers of Snort, also release pre-built packages of Snort that might be useful to you. To get the latest version of Snort, point your browser to:

http://www.snort.org/snort-downloads

As of this writing, Snort version 2.8.6.1 is the 'latest'. I'll be using this version as the base of my wireless IDS system. By the time you read this, this may be considered an older version of Snort. While that may be the case, the premise of the article should stand. Simply use the latest version of Snort.

Below is my "./configure" flags I use to build Snort.

```
./configure --prefix=/usr --mandir=/usr/share/man --infodir=/usr/share/info --datadir=/usr/share --sysconfdir=/etc --
localstatedir=/var/lib --enable-shared --disable-static --enable-dynamicplugin --disable-ipv6 --disable-gre --disable-mpls --enable-
targetbased --enable-decoder-preprocessor-rules --disable-ppm --disable-timestats --enable-perfprofiling --disable-linux-smp-stats --
enable-inline --enable-inline-init-failopen --disable-prelude --disable-debug --enable-reload --enable-reload-error-restart --disable-
flexresp --enable-flexresp2 --disable-react --disable-aruba --enable-zlib --without-mysql --without-odbc --without-postgresql --disable-
build-dynamic-examples --disable-corefiles --disable-ipfw --disable-profile --disable-ppm-test --without-oracle --enable-pthread --with-
libipq-includes=/usr/include/libipq
```

You might notice that I've disabled all SQL database support. This might seem incorrect, but it's actually just what I want. We'll get to that a bit later in the 'Barnyard' configuration section of this paper. I also enabled options like 'inline' with Snort, in the event I wish to turn our IDS system into an IPS (Intrusion Prevention System). For the time being, we're only concentrating on IDS, but having the ability to go full IPS in the future is a nice option to have.

You'll also want to create the user 'snort' on your system. While we will start Snort as the user 'root', when Snort finishes initializing, we'll want it to drop its privileges and 'become' the user name 'snort'. We do this for security reasons (ie - privilege separation). On most modern systems, when you add the 'snort' user, the account is 'disabled'. That is, it hasn't been assigned a password and thus can't be logged into. I typically like to keep it that way.

For Snort rule sets, I'll be using the Sourcefire Snort rules and Emerging threats rules. You'll need to download those and put them in the /etc/snort directory. If you wish to put your rules in another directory, don't forget to change the "RULE_PATH" in the configuration files.

The first interface we'll tackle is the Internet facing one (eth1). This is the Interface that connects our wireless AP/IDS system to the Internet. Here's my base configuration file used for that externally facing, Internet connected interface.

```
#############################################################################
# Snort configuration for the Internet facing side of our Wireless IDS/IPS #
# sensor.                                                                   #
#############################################################################


var HOME_NET [YOURIPADDRESS/32]  # Public IP address (Internet)
var EXTERNAL_NET !$HOME_NET
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 SSH_PORTS 22

portvar HTTP_PORTS  [80,2301,3128,7777,7779,8000,8008,8028,8080,8180,8888,9999]

# SMTP and HTTP trigger many false positive "shell code" alerts.  Ignore these
# ports.

portvar SHELLCODE_PORTS [!25,!80]

# Not running any Oracle servers,  but some emerging rules (exploit) make
# reference to this variable.

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,
205.188.153.0/24,205.188.179.0/24,205.188.248.0/24]
```

```
var RULE_PATH /etc/snort
var SO_RULE_PATH /etc/snort/so_rules
var PREPROC_RULE_PATH /etc/snort/preproc_rules

# Reduce the noise.

config disable_decode_alerts
config disable_tcpopt_experimental_alerts
config disable_tcpopt_obsolete_alerts
config disable_tcpopt_ttcp_alerts
config disable_tcpopt_alerts
config disable_ipopt_alerts
config checksum_mode: none

# Ignore the OpenVPN UDP port.

config ignore_ports: udp 2600

config pcre_match_limit: 1500
config pcre_match_limit_recursion: 1500

config detection: search-method ac-bnfa max_queue_events 5
config event_queue: max_queue 8 log 3 order_events content_length

# Where to find dynamic preprocessors,  libraries, etc.

dynamicpreprocessor directory /usr/lib/snort_dynamicpreprocessor/
dynamicengine /usr/lib/snort_dynamicengine/libsf_engine.so
dynamicdetection directory /usr/lib/snort_dynamicrules

preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy windows timeout 180
preprocessor stream5_global: max_tcp 8192, track_tcp yes, track_udp no
preprocessor stream5_tcp: policy windows, use_static_footprint_sizes, ports
client 21 22 23 25 42 53 79 80 109 110 111 113 119 135 136 137 139 143 110
111 161 445 513 514 691 1433 1521 2100 2301 3128 3306 6665 6666 6667 6668
6669 7000 8000 8080 8180 8888 32770 32771 32772 32773 32774 32775 32776
32777 32778 32779, ports both 443 465 563 636 989 992 993 994 995 7801
7702 7900 7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911 7912
7913 7914 7915 7916 7917 7918 7919 7920

preprocessor perfmonitor: time 300 file /var/snort/snort.stats pktcnt 10000

# We don't run any internal web servers that are connectable via the
# Internet.  However,  we'll leave this preprocesssor enabled.
#
preprocessor http_inspect: global iis_unicode_map unicode.map 1252
preprocessor http_inspect_server: server default \
    apache_whitespace no \
    ascii no \
 bare_byte no \
 chunk_length 500000 \
 flow_depth 1460 \
 directory no \
 double_decode no \
 iis_backslash no \
 iis_delimiter no \
 iis_unicode no \
 multi_slash no \
 non_strict \
 oversize_dir_length 2048 \
 ports { 80 2301 3128 7777 7779 8000 8008 8028 8080 8180 8888 9999 } \
 u_encode yes \
 non_rfc_char { 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 } \
 webroot no

preprocessor rpc_decode: 111 32770 32771 32772 32773 32774 32775 32776 32777 32778 32779
no_alert_multiple_requests no_alert_large_fragments no_alert_incomplete

preprocessor bo

preprocessor ftp_telnet: global encrypted_traffic yes check_encrypted inspection_type stateful
preprocessor ftp_telnet_protocol: telnet \
    ayt_attack_thresh 20 \
    normalize ports { 23 } \
    detect_anomalies
preprocessor ftp_telnet_protocol: ftp server default \
    def_max_param_len 100 \
    ports { 21 2100 } \
    ftp_cmds { USER PASS ACCT CWD SDUP SMNT QUIT REIN PORT PASV TYPE STRU MODE } \
    ftp_cmds { RETR STOR STOU APPE ALLO REST RNFR RNTO ABOR DELE RMD MKD PWD } \
    ftp_cmds { LIST NLST SITE SYST STAT HELP NOOP } \
    ftp_cmds { AUTH ADAT PROT PBSZ CONF ENC } \
    ftp_cmds { FEAT OPTS CEL CMD MACB } \
    ftp_cmds { MDTM REST SIZE MLST MLSD } \
    ftp_cmds { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    alt_max_param_len 0 { CDUP QUIT REIN PASV STOU ABOR PWD SYST NOOP } \
    alt_max_param_len 100 { MDTM CEL XCWD SITE USER PASS REST DELE RMD SYST TEST STAT MACB EPSV CLNT LPRT } \
```

```
    alt_max_param_len 200 { XMKD NLST ALLO STOU APPE RETR STOR CMD RNFR HELP } \
    alt_max_param_len 256 { RNTO CWD } \
    alt_max_param_len 400 { PORT } \
    alt_max_param_len 512 { SIZE } \
    chk_str_fmt { USER PASS ACCT CWD SDUP SMNT PORT TYPE STRU MODE } \
    chk_str_fmt { RETR STOR STOU APPE ALLO REST RNFR RNTO DELE RMD MKD } \
    chk_str_fmt { LIST NLST SITE SYST STAT HELP } \
    chk_str_fmt { AUTH ADAT PROT PBSZ CONF ENC } \
    chk_str_fmt { FEAT OPTS CEL CMD } \
    chk_str_fmt { MDTM REST SIZE MLST MLSD } \
    chk_str_fmt { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    cmd_validity MODE < char ASBCZ > \
    cmd_validity STRU < char FRP > \
    cmd_validity ALLO < int [ char R int ] > \
    cmd_validity TYPE < { char AE [ char NTC ] | char I | char L [ number ] } > \
    cmd_validity MDTM < [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string > \
    cmd_validity PORT < host_port >
preprocessor ftp_telnet_protocol: ftp client default \
    max_resp_len 256 \
    bounce yes \
    telnet_cmds no

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 ssh: server_ports { 22 } \
                  max_client_bytes 19600 \
                  max_encrypted_packets 20 \
                  enable_respoverflow enable_ssh1crc32 \
                  enable_srvoverflow enable_protomismatch

preprocessor dcerpc2: memcap 102400, events [co ]
preprocessor dcerpc2_server: default, policy WinXP, \
    detect [smb [139,445], tcp 135, udp 135, rpc-over-http-server 593], \
    autodetect [tcp 1025:, udp 1025:, rpc-over-http-server 1025:], \
    smb_max_chain 3

preprocessor dns: ports { 53 } enable_rdata_overflow

preprocessor ssl: ports { 443 465 563 636 989 992 993 994 995 7801 7702
7900 7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911 7912 7913
7914 7915 7916 7917 7918 7919 7920 }, trustservers, noinspect_encrypted

# Output formats.  We'll be using 'unified2' for back end logging to our
# SQL database.  Syslog and tcpdump I'm simply using as fall back recording
# methods.

output alert_syslog: LOG_AUTH LOG_ALERT
output log_tcpdump: tcpdump.eth1.log
output unified2: filename snort.eth1.log, limit 128

#############################################################################
# Rule loading.
#############################################################################

include classification.config
include reference.config

include $PREPROC_RULE_PATH/preprocessor.rules
include $PREPROC_RULE_PATH/decoder.rules

include $SO_RULE_PATH/bad-traffic.rules
include $SO_RULE_PATH/dos.rules
include $SO_RULE_PATH/exploit.rules

include $RULE_PATH/threshold.conf

include $RULE_PATH/attack-responses.rules
include $RULE_PATH/backdoor.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/blacklist.rules
include $RULE_PATH/botnet-cnc.rules
include $RULE_PATH/ddos.rules
include $RULE_PATH/emerging-attack_response.rules
include $RULE_PATH/emerging-botcc.rules
include $RULE_PATH/emerging-compromised.rules
include $RULE_PATH/emerging-current_events.rules
include $RULE_PATH/emerging-drop.rules
include $RULE_PATH/emerging-dshield.rules
include $RULE_PATH/emerging-exploit.rules
include $RULE_PATH/emerging-misc.rules
include $RULE_PATH/emerging-rbn.rules
include $RULE_PATH/emerging-scan.rules
```

```
include $RULE_PATH/emerging-shellcode.rules
include $RULE_PATH/emerging-tor.rules
include $RULE_PATH/emerging-trojan.rules
include $RULE_PATH/exploit.rules
include $RULE_PATH/scan.rules
include $RULE_PATH/shellcode.rules
include $RULE_PATH/specific-threats.rules
```

For the external interface, I'm not going to load very many rules. We have some general rules loaded to detect certain things, but the fact is, my wireless IDS system is **not** going to be running many services that will be connectable from the outside. Let me rephrase that. My wireless IDS system **will not** be running **any** publicly connectable services on the Internet facing interface. Period. With proper firewalling in place and not running un-needed services, this dramatically reduces our threat level and hence reduces the number of rules I'll be using on this interface.

Think of it this way. I'm not, and will never be running a service on the wireless IDS system that requires "Coldfusion" support. It will never be protecting anything that requires Coldfusion support. Therefore, there's not much of a reason to load the "web-coldfusion.rules" rule set!

The 'wireless' interface (wlan0) is a bit different. I will be monitoring that interface for everything from attacks to malware. Anything in our wireless network, I'm going to consider potentially hostile. The interface configuration is pretty similar, but with different network definitions and more rules loaded.

```
#############################################################################
# Internal network/wireless side of our network.   This will ultimately
# operating on our wlan0 devices (Atheros wireless card)
#############################################################################

var HOME_NET [10.220.0.0/24]  # My home network.  Alter to yours!
var EXTERNAL_NET !$HOME_NET
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 SSH_PORTS 22

portvar HTTP_PORTS  [80,2301,3128,7777,7779,8000,8008,8028,8080,8180,8888,9999]

# SMTP and HTTP trigger many false positive "shell code" alerts.  Ignore these
# ports.

portvar SHELLCODE_PORTS [!25,!80]

# Not running any Oracle servers,  but some emerging rules (exploit) make
# reference to this variable.

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,
205.188.153.0/24,205.188.179.0/24,205.188.248.0/24]

var RULE_PATH /etc/snort
var SO_RULE_PATH /etc/snort/so_rules
var PREPROC_RULE_PATH /etc/snort/preproc_rules

# Reduce the noise.

config disable_decode_alerts
config disable_tcpopt_experimental_alerts
config disable_tcpopt_obsolete_alerts
config disable_tcpopt_ttcp_alerts
config disable_tcpopt_alerts
config disable_ipopt_alerts
config checksum_mode: none

config pcre_match_limit: 1500
config pcre_match_limit_recursion: 1500

config detection: search-method ac-bnfa max_queue_events 5
config event_queue: max_queue 8 log 3 order_events content_length

# Where to find dynamic preprocessors,  libraries, etc.

dynamicpreprocessor directory /usr/lib/snort_dynamicpreprocessor/
dynamicengine /usr/lib/snort_dynamicengine/libsf_engine.so
dynamicdetection directory /usr/lib/snort_dynamicrules

preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy windows timeout 180
preprocessor stream5_global: max_tcp 8192, track_tcp yes, track_udp no
preprocessor stream5_tcp: policy windows, use_static_footprint_sizes,
ports client 21 22 23 25 42 53 79 80 109 110 111 113 119 135 136 137 139 143
110 111 161 445 513 514 691 1433 1521 2100 2301 3128 3306 6665 6666 6667 6668
6669 7000 8000 8080 8180 8888 32770 32771 32772 32773 32774 32775 32776 32777
32778 32779, ports both 443 465 563 636 989 992 993 994 995 7801 7702 7900
7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911 7912 7913 7914 7915
7916 7917 7918 7919 7920
```

```
preprocessor perfmonitor: time 300 file /var/snort/snort.stats pktcnt 10000

# We don't run any internal web servers that are connectable via the
# Internet.  However,  we'll leave this preprocesssor enabled.
#
preprocessor http_inspect: global iis_unicode_map unicode.map 1252
preprocessor http_inspect_server: server default \
    apache_whitespace no \
    ascii no \
 bare_byte no \
 chunk_length 500000 \
 flow_depth 1460 \
 directory no \
 double_decode no \
 iis_backslash no \
 iis_delimiter no \
 iis_unicode no \
 multi_slash no \
 non_strict \
 oversize_dir_length 2048 \
 ports { 80 2301 3128 7777 7779 8000 8008 8028 8080 8180 8888 9999 } \
 u_encode yes \
 non_rfc_char { 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 } \
 webroot no

preprocessor rpc_decode: 111 32770 32771 32772 32773 32774 32775 32776 32777 32778 32779
no_alert_multiple_requests no_alert_large_fragments no_alert_incomplete

preprocessor bo

preprocessor ftp_telnet: global encrypted_traffic yes check_encrypted inspection_type stateful
preprocessor ftp_telnet_protocol: telnet \
    ayt_attack_thresh 20 \
    normalize ports { 23 } \
    detect_anomalies
preprocessor ftp_telnet_protocol: ftp server default \
    def_max_param_len 100 \
    ports { 21 2100 } \
    ftp_cmds { USER PASS ACCT CWD SDUP SMNT QUIT REIN PORT PASV TYPE STRU MODE } \
    ftp_cmds { RETR STOR STOU APPE ALLO REST RNFR RNTO ABOR DELE RMD MKD PWD } \
    ftp_cmds { LIST NLST SITE SYST STAT HELP NOOP } \
    ftp_cmds { AUTH ADAT PROT PBSZ CONF ENC } \
    ftp_cmds { FEAT OPTS CEL CMD MACB } \
    ftp_cmds { MDTM REST SIZE MLST MLSD } \
    ftp_cmds { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    alt_max_param_len 0 { CDUP QUIT REIN PASV STOU ABOR PWD SYST NOOP } \
    alt_max_param_len 100 { MDTM CEL XCWD SITE USER PASS REST DELE RMD SYST TEST STAT MACB EPSV CLNT LPRT } \
    alt_max_param_len 200 { XMKD NLST ALLO STOU APPE RETR STOR CMD RNFR HELP } \
    alt_max_param_len 256 { RNTO CWD } \
    alt_max_param_len 400 { PORT } \
    alt_max_param_len 512 { SIZE } \
    chk_str_fmt { USER PASS ACCT CWD SDUP SMNT PORT TYPE STRU MODE } \
    chk_str_fmt { RETR STOR STOU APPE ALLO REST RNFR RNTO DELE RMD MKD } \
    chk_str_fmt { LIST NLST SITE SYST STAT HELP } \
    chk_str_fmt { AUTH ADAT PROT PBSZ CONF ENC } \
    chk_str_fmt { FEAT OPTS CEL CMD } \
    chk_str_fmt { MDTM REST SIZE MLST MLSD } \
    chk_str_fmt { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    cmd_validity MODE < char ASBCZ > \
    cmd_validity STRU < char FRP > \
    cmd_validity ALLO < int [ char R int ] > \
    cmd_validity TYPE < { char AE [ char NTC ] | char I | char L [ number ] } > \
    cmd_validity MDTM < [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string > \
    cmd_validity PORT < host_port >
preprocessor ftp_telnet_protocol: ftp client default \
    max_resp_len 256 \
    bounce yes \
    telnet_cmds no

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 ssh: server_ports { 22 } \
                  max_client_bytes 19600 \
                  max_encrypted_packets 20 \
                  enable_respoverflow enable_ssh1crc32 \
                  enable_srvoverflow enable_protomismatch

preprocessor dcerpc2: memcap 102400, events [co ]
preprocessor dcerpc2_server: default, policy WinXP, \
    detect [smb [139,445], tcp 135, udp 135, rpc-over-http-server 593], \
    autodetect [tcp 1025:, udp 1025:, rpc-over-http-server 1025:], \
    smb_max_chain 3
```

```
preprocessor dns: ports { 53 } enable_rdata_overflow

preprocessor ssl: ports { 443 465 563 636 989 992 993 994 995 7801
7702 7900 7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911
7912 7913 7914 7915 7916 7917 7918 7919 7920 }, trustservers,
noinspect_encrypted

# Output formats.  We'll be using 'unified2' for back end logging to our
# SQL database.  Syslog and tcpdump I'm simply using as fall back recording
# methods.

output alert_syslog: LOG_AUTH LOG_ALERT
output log_tcpdump: tcpdump.wlan0.log
output unified2: filename snort.wlan0.log, limit 128

###################################################################################
# Rule loading.
###################################################################################

include classification.config
include reference.config

include $PREPROC_RULE_PATH/preprocessor.rules
include $PREPROC_RULE_PATH/decoder.rules

include $SO_RULE_PATH/bad-traffic.rules
include $SO_RULE_PATH/dos.rules
include $SO_RULE_PATH/exploit.rules

include $RULE_PATH/threshold.conf

include $RULE_PATH/attack-responses.rules
include $RULE_PATH/backdoor.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/blacklist.rules
include $RULE_PATH/botnet-cnc.rules
include $RULE_PATH/chat.rules
include $RULE_PATH/content-replace.rules
include $RULE_PATH/ddos.rules
include $RULE_PATH/dns.rules
include $RULE_PATH/dos.rules
include $RULE_PATH/emerging-activex.rules
include $RULE_PATH/emerging-attack_response.rules
include $RULE_PATH/emerging-botcc.rules
include $RULE_PATH/emerging-chat.rules
include $RULE_PATH/emerging-compromised.rules
include $RULE_PATH/emerging-current_events.rules
include $RULE_PATH/emerging-dns.rules
include $RULE_PATH/emerging-dos.rules
include $RULE_PATH/emerging-drop.rules
include $RULE_PATH/emerging-dshield.rules
include $RULE_PATH/emerging-exploit.rules
include $RULE_PATH/emerging-ftp.rules
include $RULE_PATH/emerging-games.rules
include $RULE_PATH/emerging-imap.rules
include $RULE_PATH/emerging-inappropriate.rules
include $RULE_PATH/emerging-malware.rules
include $RULE_PATH/emerging-misc.rules
include $RULE_PATH/emerging-p2p.rules
include $RULE_PATH/emerging-policy.rules
include $RULE_PATH/emerging-pop3.rules
include $RULE_PATH/emerging-rbn.rules
include $RULE_PATH/emerging-rpc.rules
include $RULE_PATH/emerging-scan.rules
include $RULE_PATH/emerging-shellcode.rules
include $RULE_PATH/emerging-smtp.rules
include $RULE_PATH/emerging-snmp.rules
include $RULE_PATH/emerging-sql.rules
include $RULE_PATH/emerging-telnet.rules
include $RULE_PATH/emerging-tftp.rules
include $RULE_PATH/emerging-tor.rules
include $RULE_PATH/emerging-trojan.rules
include $RULE_PATH/emerging-user_agents.rules
include $RULE_PATH/emerging-virus.rules
include $RULE_PATH/emerging-voip.rules
include $RULE_PATH/emerging-web_client.rules
include $RULE_PATH/emerging-web_server.rules
include $RULE_PATH/emerging-web_specific_apps.rules
include $RULE_PATH/emerging-worm.rules
include $RULE_PATH/experimental.rules
include $RULE_PATH/exploit.rules
include $RULE_PATH/finger.rules
include $RULE_PATH/ftp.rules
include $RULE_PATH/imap.rules
include $RULE_PATH/info.rules
include $RULE_PATH/local.rules
include $RULE_PATH/misc.rules
include $RULE_PATH/multimedia.rules
```

```
include $RULE_PATH/mysql.rules
include $RULE_PATH/netbios.rules
include $RULE_PATH/nntp.rules
include $RULE_PATH/oracle.rules
include $RULE_PATH/other-ids.rules
include $RULE_PATH/p2p.rules
include $RULE_PATH/phishing-spam.rules
include $RULE_PATH/policy.rules
include $RULE_PATH/pop3.rules
include $RULE_PATH/rpc.rules
include $RULE_PATH/rservices.rules
include $RULE_PATH/scada.rules
include $RULE_PATH/scan.rules
include $RULE_PATH/shellcode.rules
include $RULE_PATH/smtp.rules
include $RULE_PATH/snmp.rules
include $RULE_PATH/specific-threats.rules
include $RULE_PATH/spyware-put.rules
include $RULE_PATH/sql.rules
include $RULE_PATH/telnet.rules
include $RULE_PATH/tftp.rules
include $RULE_PATH/virus.rules
include $RULE_PATH/voip.rules
include $RULE_PATH/web-activex.rules
include $RULE_PATH/web-attacks.rules
include $RULE_PATH/web-cgi.rules
include $RULE_PATH/web-client.rules
include $RULE_PATH/web-coldfusion.rules
include $RULE_PATH/web-frontpage.rules
include $RULE_PATH/web-iis.rules
include $RULE_PATH/web-misc.rules
include $RULE_PATH/web-php.rules
```

As you can see, many more rules have been loaded. In reality, you could probably trim down the rule sets loaded a good bit. However, this is a PoC model, so I'm pretty much "throwing the book" at it.

To start and test your Snort configuration, simply run Snort with the appropriate flags:

```
# snort -i wlan0 -c /etc/snort/snort.wlan0.conf -u snort -g snort
```

Substitute the interface (-i wlan0) and configuration file (snort.wlan0.conf) with the appropriate settings for your setup. For this article, I'm simply giving a brief run down of my configurations. Your configuration may vary. If you have problems with Snort, Google around. There are many "HOWTO" documents related to Snort and this article isn't really geared toward a person who is completely new to Snort. Once you verified the Snort setup, you can run Snort with the same configuration options, but add a '-D' (daemonize). This will make Snort run in the background. You might want to consider adding Snort to your system's startup routine. Make sure you put it in **after** your wireless and hostapd are set up!

It's probably not apparent yet, but we have one more interface we'll be running Snort on. This interface will be a TAP/TUN device supplied to us by Kismet.

Kismet monitoring with Snort in the mix

"Kismet is an 802.11 layer2 wireless network detector, sniffer, and intrusion detection system. Kismet will work with any wireless card which supports raw monitoring (rfmon) mode, and (with appropriate hardware) can sniff 802.11b, 802.11a, 802.11g, and 802.11n traffic. Kismet also supports plug-ins which allow sniffing other media such as DECT. Kismet identifies networks by passively collecting packets and detecting standard named networks, detecting (and given time, decloaking) hidden networks, and inferring the presence of non-beaconing networks via data traffic." -- From the Kismet web site

Remember, we have *two* Atheros based cards. We're using one card (wlan0) to create our "Access Point" for internal users to connect to the network.

We're already monitoring that network interface with Snort, as well as our network interface connecting our Wireless IDS system to the Internet.

The second wireless interface (wlan1) will be our "Kismet" interface. Kismet will actually supply us with two sets of data. The first set of data supplied by Kismet is layer 2 data. That data will be Kismet scanning wireless frequencies looking for new clients, potentially hostile/rogue AP, and miscellaneous other layer 2 types of attacks.

Basically, we'll be operating Kismet in a "traditional" sense. Think of "war driving", but of course our sensor won't be moving.

Kismet also has a mechanism to supply data collected (packets), as it is channel 'hopping' to a TAP/TUN interface. Think of it this way, as Kismet hops from channel to channel, it will sometimes intercept data (TCP/IP packets). Those data packets can be read by programs like tcpdump or wireshark from the Kismet supplied TAP/TUN interface.

This TAP/TUN interface will be "monitored" by our third Snort sensor.

As of this writing, the 'latest' version of Kismet is "Kismet-2010-07-R1" and can be found at http://www.kismetwireless.net. However, we'll be using Kismet with a bit of a twist to it. In our case, we want the information that Kismet "finds" via "channel hopping" to be fed to syslog for later examination. The reason for this will become more clear later in the article, but we'll go ahead and "patch" Kismet now.

You can download the Kismet "syslog" patch from:

http://sagan.quadrantsec.com/patches/kismet-2010-07-R1-syslog-patch-r4.diff

This will only patch the "kismet_server.cc" file. Once you've downloaded Kismet-2010-07-R1 and the kismet-2010-07-R1-syslog-patch-r4.diff patch, application and configuration is pretty simple. For example:

```
$ tar -zxvpf kismet-2010-07-R1.tar.gz
```

```
$ patch -p0 < kismet-2010-07-R1-syslog-patch-r3.diff
patching file kismet-2010-07-R1/kismet_server.cc
$ cd kismet-2010-07-R1
$ ./configure
$ make dep
$ make
$ sudo make install
$ sudo useradd kismet
```

Configuration is pretty simple for Kismet. It should be noted that we will *not* be using the Kismet console! We'll be using the Kismet server, and passing that information to our back end MySQL/Console system. That's not to say you *can't* use the Kismet console. Since the 'kismet_server' will be running on remote wireless IDS sensors, you can fire up the Kismet client/console *on the sensors*. What we are going for is having the 'kismet_server' report information to our back end. This means the console isn't as important to us. I'll go into my reasoning for this later in the document.

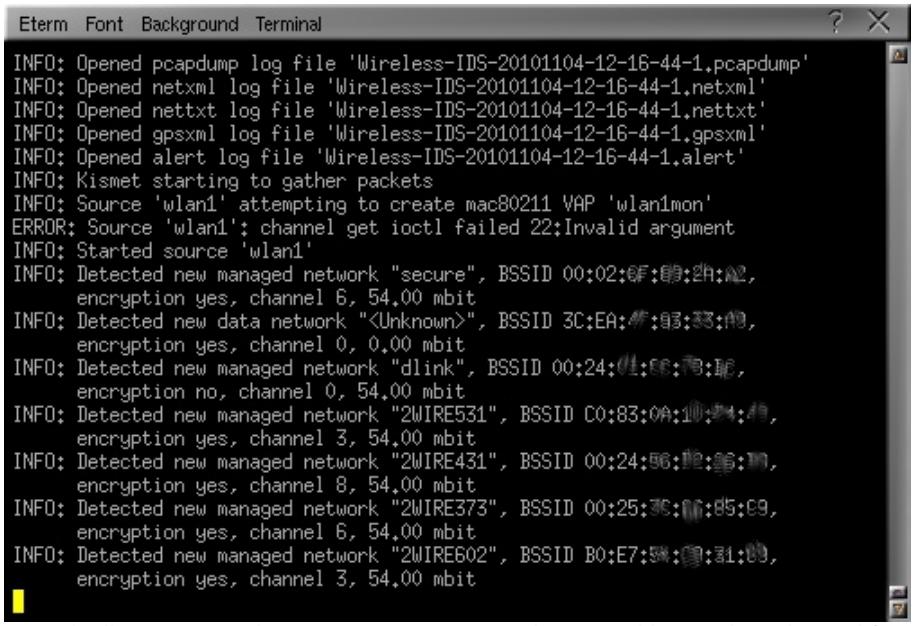
Below is my Kismet configuration (/usr/local/etc/kismet.conf).

```
################################################################################
# kismet.conf - Stripped down Kismet configuration file.  This will be read #
# by our /usr/local/bin/kismet_server                                          #
################################################################################

version=2009-newcore
servername=Kismet_2009

allowplugins=true

# Our source will be our secondary Atheros wireless card [wlan1].  Remember,
# our first card (wlan0) is operating as our AP.

ncsource=wlan1

# Common channels,  and how often to "hop" (1 channel per second)

preferredchannels=1,6,11
channelvelocity=1

# Channel list supported.  This must be present for Kismet to operate!

channellist=IEEE80211b:1:3,6:3,11:3,2,7,3,8,4,9,5,10
channellist=IEEE80211a:36,40,44,48,52,56,60,64,149,153,157,161,165
channellist=IEEE80211ab:1:3,6:3,11:3,2,7,3,8,4,9,5,10,36,40,44,48,52,56,60,64,149,153,157,161,165

# We don't be using this,  but kismet_server won't start without it.

listen=tcp://127.0.0.1:2501
allowedhosts=127.0.0.1
maxbacklog=5000

# If you have Wireshark installed,  you can tell Kismet to report the
# manufacturer of a device based on the MAC address.   You may need to
# adjust this to your installation.

ouifile=/usr/share/wireshark/manuf

# This will be a stationary system.  No need for GPS.

gps=false

# Kismet will supply a TAP/TUN device.  We'll have our third instance of
# Snort running on this interface.   Make sure you name the "device"
# correctly (kistap0 is probably already taken by our AP instance)

tuntap_export=true
tuntap_device=kistap1

# Intrusion Detection Controls.  This is the backbone or our layer 2
# wireless IDS system.

alert=ADHOCCONFLICT,5/min,1/sec
alert=AIRJACKSSID,5/min,1/sec
alert=APSPOOF,10/min,1/sec
alert=BCASTDISCON,5/min,2/sec
alert=BSSTIMESTAMP,5/min,1/sec
alert=CHANCHANGE,5/min,1/sec
alert=CRYPTODROP,5/min,1/sec
alert=DISASSOCTRAFFIC,10/min,1/sec
alert=DEAUTHFLOOD,5/min,2/sec
alert=DEAUTHCODEINVALID,5/min,1/sec
alert=DISCONCODEINVALID,5/min,1/sec
alert=DHCPNAMECHANGE,5/min,1/sec
alert=DHCPOSCHANGE,5/min,1/sec
alert=DHCPCLIENTID,5/min,1/sec
alert=DHCPCONFLICT,10/min,1/sec
alert=NETSTUMBLER,5/min,1/sec
alert=LUCENTTEST,5/min,1/sec
alert=LONGSSID,5/min,1/sec
alert=MSFBCOMSSID,5/min,1/sec
alert=MSFDLINKRATE,5/min,1/sec
```

```
alert=MSFNETGEARBEACON,5/min,1/sec
alert=NULLPROBERESP,5/min,1/sec
alert=PROBENOJOIN,5/min,1/sec

# This is the SSID of our AP (wlan0; Atheros card) with that card's
# MAC address.  We set this to detect attackers attempting to "spoof" our
# AP (hopefully).

apspoof=AP:ssid="secure",validmacs="00:02:6f:89:2a:a2"

# How often to write files.  0 will disable it.  However,  I let Kismet
# write its own files.  This way,  I always have a "backup" method of
# retrieving IDS data.

writeinterval=300

# If a tree falls in the woods,  and nobody is around......

enablesound=false
enablespeech=false

# Formats the Kismet server to "save" information.  I leave this default.
# (see "writeinterval" for my reasoning).

logtypes=pcapdump,gpsxml,netxml,nettxt,alert
pcapdumpformat=ppi

# What to pre-pend file named with...

logdefault=Wireless-IDS
logtemplate=%p%n-%D-%t-%i.%l
configdir=%h/.kismet/
```

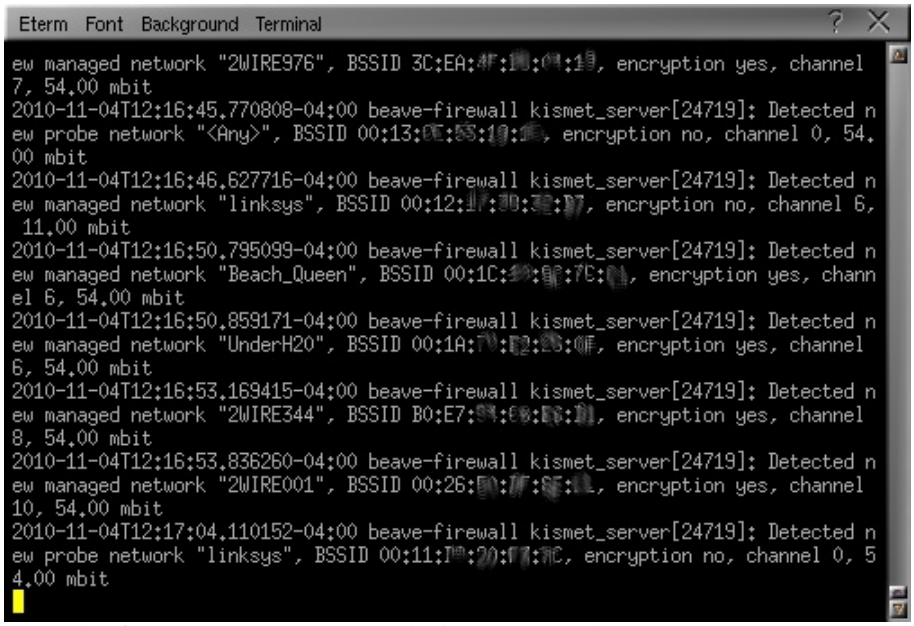
Note: Be sure to change the 'apspoof' option to the MAC address of your wireless AP card (wlan0)! This way, Kismet can monitor the "out side" for possible AP spoofing attempts!

After reviewing and adjusting your configurations, you can test by firing up /usr/local/bin/kismet_server. With any luck, you'll see something like this:



(Figure 3. The 'kismet_server' running in interactive mode. MAC addresses have been obfuscated)

While the Kismet server is up and running, it might be a good time to check your syslog information. On my PoC system, you'd type, 'tail -f /var/log/messages'.



(Figure 4. Example of syslog output supplied by the 'kismet_server'. MAC addresses have been obfuscated)

You might need to check your system/distribution setup for where the syslog information gets stored.

Once you're satisfied with the Kismet server and syslog results, we can go ahead and start up the Kismet server in the background. This can be done by running the /usr/local/bin/kismet_server with the --daemonize flag. To see other kismet_server flags, pass the --help option.

With the 'kismet_server' operational, we should have a new pseudo interface named "kistap1". This interface is being supplied by Kismet, and this is the data (packet level) that's being gathered while it does its 'channel hopping'.

Note: 'mmiller' on irc.freenode.net in the #kismet channel pointed me to the Janus Project. This project makes use of multiple wifi cards to monitor all possible wireless channels. 'mmiller' pointed out that the best way to monitor the wireless data intercepted on other channels outside of my network would be to do it this way. The reasoning is that since Kismet is only intercepting data on one channel at a time, I'd be missing data on other channels. I believe he is correct, however, this ended up outside of the scope of this PoC project.

We'll be running Snort on this interface. Since it's the "channel hopping" interface, things get a little more strange. For example, we have clear needs and solutions for all of our Snort interfaces. For example, Snort is monitoring our externally facing interface for external attacks (eth1). We also monitor our internal interface with Snort for attacks and malicious traffic from within our network at the AP level (wlan0). This interface doesn't really "fit" into those categories.

So basically, this interface is going to be monitoring 'data' *outside* of our network. This is essentially what Google got in trouble for! If a nearby network isn't encrypted and Kismet "picks up" some of that packet data, we'll see it on our kistap1 interface. So, essentially, we're running Snort on "outside of our network" traffic. We're grabbing packets out of the air an analyzing them.

Do we need to? Probably not. Since this is a PoC design, I'm interested in seeing the data that's supplied. Some might even consider this a slippery legal issue.

In a real world design, this might be over the limit. I'll leave that to the reader. However, the beer is telling me this is a "research project", so let's continue forward.

Considering we don't really have a clear idea what type of traffic we'll be seeing on this interface, we'll use a similar Snort configuration that we used on our AP (wlan0). That is, we'll enable lots of rules and see what it picks up. Unlike previous configurations, where we had a clear definition of TCP/IP address ranges that would be in use, on this interface we do not.

The packets received by our Kismet supplied interface could be within any possible range. Some might be RFC1918, while others might not be. To get around this, we want to run Snort in an "any/any" configuration. That is, any traffic coming from/to any place needs to be examined. However, when we tell Snort to run in a "HOME_NET any" and "EXTERNAL_NET any" configuration, certain rules will complain (mostly emerging threat rules). To get around this issue, I'm creating a 'mirror' of our rules in the "/etc/snort/any" directory. Rules that complain about our 'any/any' configuration will be disabled. It's typically only a handful of rules that cause this problem. By doing it this way, we can keep our kistap1 rules separate from our "real rules".

This is basically the run-down of my Snort configuration file that we're going to be using for our kistap1 interface. Mine is stored in the /etc/snort/snort.kistap1.conf.

```
#############################################################################
# Snort configuration for our 'channel hopping' interface supplied by
# Kismet [/etc/snort/snort.kistap1.conf]
#############################################################################

# This will break some rules (any/any) so be prepared to correct manually
# and/or with oinkmaster/pulledpork.

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 SSH_PORTS 22

portvar HTTP_PORTS  [80,2301,3128,7777,7779,8000,8008,8028,8080,8180,8888,9999]

# SMTP and HTTP trigger many false positive "shell code" alerts.  Ignore these
# ports.

portvar SHELLCODE_PORTS [!25,!80]

# Not running any Oracle servers,  but some emerging rules (exploit) make
# reference to this variable.

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,
205.188.153.0/24,205.188.179.0/24,205.188.248.0/24]

var RULE_PATH /etc/snort/any
var SO_RULE_PATH /etc/snort/so_rules
var PREPROC_RULE_PATH /etc/snort/preproc_rules

# Reduce the noise.

config disable_decode_alerts
config disable_tcpopt_experimental_alerts
config disable_tcpopt_obsolete_alerts
config disable_tcpopt_ttcp_alerts
config disable_tcpopt_alerts
config disable_ipopt_alerts
config checksum_mode: none

config pcre_match_limit: 1500
config pcre_match_limit_recursion: 1500

config detection: search-method ac-bnfa max_queue_events 5
config event_queue: max_queue 8 log 3 order_events content_length

# Where to find dynamic preprocessors,  libraries, etc.

dynamicpreprocessor directory /usr/lib/snort_dynamicpreprocessor/
dynamicengine /usr/lib/snort_dynamicengine/libsf_engine.so
dynamicdetection directory /usr/lib/snort_dynamicrules

preprocessor frag3_global: max_frags 65536
preprocessor frag3_engine: policy windows timeout 180
preprocessor stream5_global: max_tcp 8192, track_tcp yes, track_udp no
preprocessor stream5_tcp: policy windows, use_static_footprint_sizes, ports
client 21 22 23 25 42 53 79 80 109 110 111 113 119 135 136 137 139 143 110
111 161 445 513 514 691 1433 1521 2100 2301 3128 3306 6665 6666 6667 6668
6669 7000 8000 8080 8180 8888 32770 32771 32772 32773 32774 32775 32776
32777 32778 32779, ports both 443 465 563 636 989 992 993 994 995 7801
7702 7900 7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911 7912
7913 7914 7915 7916 7917 7918 7919 7920

preprocessor perfmonitor: time 300 file /var/snort/snort.stats pktcnt 10000

# We don't run any internal web servers that are connectable via the
# Internet.  However,  we'll leave this preprocesssor enabled.
#
preprocessor http_inspect: global iis_unicode_map unicode.map 1252
preprocessor http_inspect_server: server default \
    apache_whitespace no \
    ascii no \
 bare_byte no \
 chunk_length 500000 \
 flow_depth 1460 \
 directory no \
 double_decode no \
 iis_backslash no \
 iis_delimiter no \
 iis_unicode no \
 multi_slash no \
 non_strict \
 oversize_dir_length 2048 \
 ports { 80 2301 3128 7777 7779 8000 8008 8028 8080 8180 8888 9999 } \
 u_encode yes \
 non_rfc_char { 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 } \
 webroot no

preprocessor rpc_decode: 111 32770 32771 32772 32773 32774 32775 32776 32777 32778 32779
no_alert_multiple_requests no_alert_large_fragments no_alert_incomplete

preprocessor bo

preprocessor ftp_telnet: global encrypted_traffic yes check_encrypted inspection_type stateful
preprocessor ftp_telnet_protocol: telnet \
    ayt_attack_thresh 20 \
```

```
    normalize ports { 23 } \
    detect_anomalies
preprocessor ftp_telnet_protocol: ftp server default \
    def_max_param_len 100 \
    ports { 21 2100 } \
    ftp_cmds { USER PASS ACCT CWD SDUP SMNT QUIT REIN PORT PASV TYPE STRU MODE } \
    ftp_cmds { RETR STOR STOU APPE ALLO REST RNFR RNTO ABOR DELE RMD MKD PWD } \
    ftp_cmds { LIST NLST SITE SYST STAT HELP NOOP } \
    ftp_cmds { AUTH ADAT PROT PBSZ CONF ENC } \
    ftp_cmds { FEAT OPTS CEL CMD MACB } \
    ftp_cmds { MDTM REST SIZE MLST MLSD } \
    ftp_cmds { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    alt_max_param_len 0 { CDUP QUIT REIN PASV STOU ABOR PWD SYST NOOP } \
    alt_max_param_len 100 { MDTM CEL XCWD SITE USER PASS REST DELE RMD SYST TEST STAT MACB EPSV CLNT LPRT } \
    alt_max_param_len 200 { XMKD NLST ALLO STOU APPE RETR STOR CMD RNFR HELP } \
    alt_max_param_len 256 { RNTO CWD } \
    alt_max_param_len 400 { PORT } \
    alt_max_param_len 512 { SIZE } \
    chk_str_fmt { USER PASS ACCT CWD SDUP SMNT PORT TYPE STRU MODE } \
    chk_str_fmt { RETR STOR STOU APPE ALLO REST RNFR RNTO DELE RMD MKD } \
    chk_str_fmt { LIST NLST SITE SYST STAT HELP } \
    chk_str_fmt { AUTH ADAT PROT PBSZ CONF ENC } \
    chk_str_fmt { FEAT OPTS CEL CMD } \
    chk_str_fmt { MDTM REST SIZE MLST MLSD } \
    chk_str_fmt { XPWD XCWD XCUP XMKD XRMD TEST CLNT } \
    cmd_validity MODE < char ASBCZ > \
    cmd_validity STRU < char FRP > \
    cmd_validity ALLO < int [ char R int ] > \
    cmd_validity TYPE < { char AE [ char NTC ] | char I | char L [ number ] } > \
    cmd_validity MDTM < [ date nnnnnnnnnnnnnn[.n[n[n]]] ] string > \
    cmd_validity PORT < host_port >
preprocessor ftp_telnet_protocol: ftp client default \
    max_resp_len 256 \
    bounce yes \
    telnet_cmds no

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 ssh: server_ports { 22 } \
                  max_client_bytes 19600 \
                  max_encrypted_packets 20 \
                  enable_respoverflow enable_ssh1crc32 \
                  enable_srvoverflow enable_protomismatch

preprocessor dcerpc2: memcap 102400, events [co ]
preprocessor dcerpc2_server: default, policy WinXP, \
    detect [smb [139,445], tcp 135, udp 135, rpc-over-http-server 593], \
    autodetect [tcp 1025:, udp 1025:, rpc-over-http-server 1025:], \
    smb_max_chain 3

preprocessor dns: ports { 53 } enable_rdata_overflow

preprocessor ssl: ports { 443 465 563 636 989 992 993 994 995 7801
7702 7900 7901 7902 7903 7904 7905 7906 6907 7908 7909 7910 7911
7912 7913 7914 7915 7916 7917 7918 7919 7920 }, trustservers,
noinspect_encrypted

# Output formats.  We'll be using 'unified2' for back end logging to our
# SQL database.  Syslog and tcpdump I'm simply using as fall back recording
# methods.

output alert_syslog: LOG_AUTH LOG_ALERT
output log_tcpdump: tcpdump.kistap1.log
output unified2: filename snort.kistap1.log, limit 128

##################################################################################
# Rule loading.
##################################################################################

include classification.config
include reference.config

include $PREPROC_RULE_PATH/preprocessor.rules
include $PREPROC_RULE_PATH/decoder.rules

include $SO_RULE_PATH/bad-traffic.rules
include $SO_RULE_PATH/dos.rules
include $SO_RULE_PATH/exploit.rules

include $RULE_PATH/threshold.conf

include $RULE_PATH/attack-responses.rules
```

```
include $RULE_PATH/backdoor.rules
include $RULE_PATH/bad-traffic.rules
include $RULE_PATH/blacklist.rules
include $RULE_PATH/botnet-cnc.rules
include $RULE_PATH/chat.rules
include $RULE_PATH/content-replace.rules
include $RULE_PATH/ddos.rules
include $RULE_PATH/dns.rules
include $RULE_PATH/dos.rules
include $RULE_PATH/emerging-activex.rules
include $RULE_PATH/emerging-attack_response.rules
include $RULE_PATH/emerging-botcc.rules
include $RULE_PATH/emerging-chat.rules
include $RULE_PATH/emerging-compromised.rules
include $RULE_PATH/emerging-current_events.rules
include $RULE_PATH/emerging-dns.rules
include $RULE_PATH/emerging-dos.rules
include $RULE_PATH/emerging-drop.rules
include $RULE_PATH/emerging-dshield.rules
include $RULE_PATH/emerging-exploit.rules
include $RULE_PATH/emerging-ftp.rules
include $RULE_PATH/emerging-games.rules
include $RULE_PATH/emerging-imap.rules
include $RULE_PATH/emerging-inappropriate.rules
include $RULE_PATH/emerging-malware.rules
include $RULE_PATH/emerging-misc.rules
#include $RULE_PATH/emerging-netbios.rules
include $RULE_PATH/emerging-p2p.rules
include $RULE_PATH/emerging-policy.rules
include $RULE_PATH/emerging-pop3.rules
include $RULE_PATH/emerging-rbn.rules
include $RULE_PATH/emerging-rpc.rules
include $RULE_PATH/emerging-scan.rules
include $RULE_PATH/emerging-shellcode.rules
include $RULE_PATH/emerging-smtp.rules
include $RULE_PATH/emerging-snmp.rules
include $RULE_PATH/emerging-sql.rules
include $RULE_PATH/emerging-telnet.rules
include $RULE_PATH/emerging-tftp.rules
include $RULE_PATH/emerging-tor.rules
include $RULE_PATH/emerging-trojan.rules
include $RULE_PATH/emerging-user_agents.rules
include $RULE_PATH/emerging-virus.rules
include $RULE_PATH/emerging-voip.rules
include $RULE_PATH/emerging-web_client.rules
include $RULE_PATH/emerging-web_server.rules
include $RULE_PATH/emerging-web_specific_apps.rules
include $RULE_PATH/emerging-worm.rules
include $RULE_PATH/experimental.rules
include $RULE_PATH/exploit.rules
include $RULE_PATH/finger.rules
include $RULE_PATH/ftp.rules
include $RULE_PATH/imap.rules
include $RULE_PATH/info.rules
include $RULE_PATH/local.rules
include $RULE_PATH/misc.rules
include $RULE_PATH/multimedia.rules
include $RULE_PATH/mysql.rules
include $RULE_PATH/netbios.rules
include $RULE_PATH/nntp.rules
include $RULE_PATH/oracle.rules
include $RULE_PATH/other-ids.rules
include $RULE_PATH/p2p.rules
include $RULE_PATH/phishing-spam.rules
include $RULE_PATH/policy.rules
include $RULE_PATH/pop3.rules
include $RULE_PATH/rpc.rules
include $RULE_PATH/rservices.rules
include $RULE_PATH/scada.rules
include $RULE_PATH/scan.rules
include $RULE_PATH/shellcode.rules
include $RULE_PATH/smtp.rules
include $RULE_PATH/snmp.rules
include $RULE_PATH/specific-threats.rules
include $RULE_PATH/spyware-put.rules
include $RULE_PATH/sql.rules
include $RULE_PATH/telnet.rules
include $RULE_PATH/tftp.rules
include $RULE_PATH/virus.rules
include $RULE_PATH/voip.rules
include $RULE_PATH/web-activex.rules
include $RULE_PATH/web-attacks.rules
include $RULE_PATH/web-cgi.rules
include $RULE_PATH/web-client.rules
include $RULE_PATH/web-coldfusion.rules
include $RULE_PATH/web-frontpage.rules
include $RULE_PATH/web-iis.rules
include $RULE_PATH/web-misc.rules
include $RULE_PATH/web-php.rules
```

We can now "test" Snort by running it like our previous interfaces. Simply type:

```
# snort -i kistap1 -c /etc/snort/snort.kistap1.conf -u snort -g snort
```

If it runs successfully, consider re-running the command with the -D (daemonize) option. You'll probably want to add this to your system rc/startup routines. Remember, *Kismet supplies this interface so you'll need to start Kismet **first** before running this instance of Snort!*.

Barnyard2 configurations

You might have noticed in our Snort configurations the lack of sending data to the MySQL (in our case) back end. There's a good reason for this. We'll be using Barnyard2 to collect data from Snort and send it to our back end data collection point. In each of our Snort configuration files, we're using the "output unified2:" option. As Snort detects potentially hostile traffic, it'll log it to a file using unitied2 format. This means, Snort will record the alert and rule that was triggered, as well as the packet information (packet dump). Barnyard2's job is to "read" this file, in real time, and relay that information to our SQL back end servers. The basic idea is to separate Snort from having to deal with the logging process.

Think of it this way. Snort is monitoring a network interface in real time. If a 'bad event' happens, Snort would have to temporarily 'stop processing' network traffic to log information to the SQL database. This is not good, as we might potentially miss other hostile traffic while Snort is dealing with logging to the SQL server. To avoid this, a separate process is started (Barnyard2) that'll take information from Snort, via the unified2 output format, and INSERT it into the SQL database for Snort; thus off loading that process from Snort to Barnyard2. Since Snort doesn't need to 'concentrate' efforts on SQL logging, it can continue monitoring the network interface.

In each Snort configuration, we've told each instance of Snort to log to its own individual "unified2" file. For example:

```
output unified2: filename snort.eth1.log, limit 128    # Internet connection
output unified2: filename snort.wlan0.log, limit 128   # AP interface
output unified3: filename snort.kistap1.log, limit 128 # Hopping interface
```

[Note: the 'limit' option means the unifed2 file size limit. In our cases, we're setting this to 128 megabytes

Since we have a total of three instances of Snort running, we'll be running three instances of Barnyard2. Each one will read a unique Snort unifed2 formated file. Each Barnyard2 instance will log to our SQL back end as a separate "sensor id". In the Snort SQL database in the 'sensor' table, the 'sensor id' is known as the 'sid'.

Data from each specific instance of Barnyard2 will be loaded to a unique sid. In the end, this allows you to specify traffic by interface from the database. For example, in the end, you'll be able to 'view' data from the sid # 1 (Internet connection) while ignoring sid's #2 (AP) and #3 (Kismet TAP/TUN). Or you can combined all that information together, to get a complete view of all data.

Let's first do a quick overview of creating the database and tables that'll be needed on our MySQL back end server. Keep in mind, I'm using OpenVPN to create a tunnel between our satellite wireless IDS systems and the back end server. The idea is that we'll be collecting information from Snort via Barnyard2 and sending that over the OpenVPN to our MySQL server.

In our case, our satellite (wireless IDS) system has an OpenVPN tunnel address of 1.1.2.80. The peer TCP/IP address on our back end data collection point it to 1.1.2.70. Lets first do a brief overview of setting up the SQL back end server. These instructions are for MySQL, however, PostgreSQL works very well with Barynard2 too.

First, on the server back end, lets create our database and build our tables. Database schemas ship with the tarballs of Snort within the 'schemas' directory. There you'll find MySQL, MS-SQL, DB2 and MySQL schemes. In our example, we'll be using the MySQL schema.

```
# mysqladmin -u root -p create snort_beave
# cd /home/beave/snort-2.8.6.1-build/snort-2.8.6.1/schemas
# mysql -u root -p snort_beave < create_mysql
```

Once the database is created, we need to assign the appropriate grants/ rights to the database for the remote satellite wireless IDS systems to be able to store information.

```
# mysql -u root -p snort_beave
mysql> grant INSERT,SELECT on snort_beave.* to snort@1.1.2.80 identified by 'mypassword';
mysql> grant INSERT,UPDATE,SELECT on snort_beave.sensor to snort@1.1.2.80 identified by 'mypassword'
mysql> exit
```

Note: In too many cases I've seen people grant "ALL" to remote satellite IDS/IPS systems. We only want to give our IDS/IPS system the rights that it needs! Also, think of it this way, if an attacker gains access to our system and thus steals our database credentials, do we really want them to have ALL access to the back end database?

With the grants/rights in place, we can now move on to configuring Barnyard2 to read our unifed2 output from Snort and inserting data into our MySQL database. Just like Snort, where we separated configuration files via interface (ie - snort.eth1.conf, snort.wlan0.conf, snort.kistap1.conf), we'll be doing the same with Barnyard2 (barnyard.eth1.conf, barnyard.wlan0.conf, barnyard.kistap1.conf).

I'll be putting all my Barnyard2 configuration files in the /etc/barnyard2 directory. So the first interface we'll tackle is eth1, our Internet facing interface. So, our configuration file will be barnyard2.eth1.conf

```
##############################################################################
# /etc/barnyard2/barnyard.eth1.conf - Internet facing interface.
##############################################################################

config reference_file: /etc/snort/reference.config
config classification_file: /etc/snort/classification.config
config gen_file: /etc/snort/gen-msg.map
config sid_file: /etc/snort/sid-msg.map

# The 'hostname' will be directly in our console,  so name it something
# descriptive.

config hostname: wireless-internet
```

```
config interface: eth1
config set_gid: snort
config set_uid: snort
config waldo_file: /var/log/barnyard2/barnyard2.eth1.waldo
input unified2

output database: log, mysql, user=snort password=mypassword dbname=snort_beave host=1.1.2.70, detail full
```

Obviously, you'll need to tweak this configuration to your environment (ie - 'mypassword').

To start Barnyard for the first time, type:

```
# /usr/bin/barnyard2 -c /etc/barnyard2/barnyard.eth1.conf -f snort.eth1.log -d /var/log/snort
-a /var/log/snort/archive -v
```

You should see output similar to the below:

```
File  Edit  View  Terminal  Help

        --== Initializing Barnyard2 ==--
Initializing Input Plugins!
Initializing Output Plugins!
Parsing config file "/etc/barnyard2/barnyard.eth1.conf"
Log directory = /var/log/barnyard2
Node unique name is: wireless-internet:eth1

database: compiled support for (mysql)
database: configured to use mysql
database: schema version = 107
database:           host = 1.1.2.70
database:           user = snort
database:  database name = snort_beave
database:    sensor name = wireless-internet:eth1
database:      sensor id = 1
database:  data encoding = hex
database:   detail level = full
database:     ignore_bpf = no
database: using the "log" facility

        --== Initialization Complete ==--

           -*> Barnyard2 <*-
 /  ,,_  \  Version 2.1.9-beta1 (Build 251)
 |o"  )~|  By the SecurixLive.com Team: http://www.securixlive.com/about.php
 + ''''  +  (C) Copyright 2008-2010 SecurixLive.

           Snort by Martin Roesch & The Snort Team: http://www.snort.org/team.htm
l
           (C) Copyright 1998-2007 Sourcefire Inc., et al.

Using waldo file '/var/log/barnyard2/barnyard2.eth1.waldo':
    spool directory = /var/log/snort
    spool filebase  = snort.eth1.log
    time_stamp      = 1287445084
    record_idx      = 5190
Opened spool file '/var/log/snort/snort.eth1.log.1287445084'
Waiting for new data
```

(Figure 5. Barnyard2 starting up. Pay attention to the 'database' information)

Other Barnyard2 configurations from the remaining interfaces will be very similar to our Internet facing side. I've including them in this document, but the changes should be pretty obvious (ie - waldo_file, hostname and interface)

```
##############################################################################
# /etc/barnyard2/barnyard2.wlan0.conf - Internal AP/wlan0 interface.
##############################################################################

config reference_file: /etc/snort/reference.config
config classification_file: /etc/snort/classification.config
config gen_file: /etc/snort/gen-msg.map
config sid_file: /etc/snort/sid-msg.map
config hostname: wireless-internal
config interface: wlan0
config set_gid: snort
config set_uid: snort
```

```
config waldo_file: /var/log/barnyard2/barnyard2.wlan0.waldo
input unified2

output database: log, mysql, user=snort password=mypassword dbname=snort_beave host=1.1.2.70, detail full
```

Command line to start Barnyard2:

```
# /usr/bin/barnyard2 -c /etc/barnyard2/barnyard.wlan0.conf -f snort.wlan0.log -d /var/log/snort
-a /var/log/snort/archive -v
```

```
##############################################################################
# /etc/barnyard2/barnyard2.kistap1.conf - Channel hopping interface.
##############################################################################

config reference_file: /etc/snort/reference.config
config classification_file: /etc/snort/classification.config
config gen_file: /etc/snort/gen-msg.map
config sid_file: /etc/snort/sid-msg.map
config hostname: wireless-internal
config interface: kistap1
config set_gid: snort
config set_uid: snort
config waldo_file: /var/log/barnyard2/barnyard2.kistap1.waldo
input unified2

output database: log, mysql, user=snort password=mypassword dbname=snort_beave host=1.1.2.70, detail full
```

Command line to start Barnyard2:

```
# /usr/bin/barnyard2 -c /etc/barnyard2/barnyard.kistap1.conf -f snort.kistap1.log -d /var/log/snort
-a /var/log/snort/archive -v
```

Pulling the 'kismet_server' into the mix

To veteran Kismet users, this setup is probably a bit strange. Why not just run Kismet "drones" on the remote wireless IDS systems? The "drones" can report back to a centralized server.

This idea is okay, but relies on the Kismet server to do the data interpretation of the drones. That is, if the Kismet server "dies", information from the drones is lost. Also, I still need a method to "push" information from the Kismet server to our MySQL back end. Kismet supports various log output formats, which is excellent. However, there is a lack of output formats that transport data over a network into, for example, a database format.

Another thought was to do multiple Kismet drone/servers in the field, but we end up with the same situation. That is, the lack of a network capable of transporting to our back end.

I brought this up to Dragorn on the #kismet IRC channel (irc.freenode.net). Dragorn's response was to build an application that'll query the Kismet API via the Kismet server and pull the information we need. Once the application has the information we need, we could transport it over the network however we see fit. Overall, I completely agree with Dragorn. That is a possible way to accomplish network transmission of data.

But, this misses the overall problem. Kismet, at its core, lacks any means to transport information over a network.

I had mentioned that basically what I wanted was the exact same information the Kismet console was being given, but transmitted over syslog. Dragorn opposed this idea, and again pointed me back to the Kismet API idea.

In some aspects, I completely agree with Dragorn. In actuality, I might be able to pull more information from the Kismet servers. However, this adds a bit more complexity to our wireless IDS solution. It could even be argued that it's not much more complexity at all. However, creating a syslog patch for the Kismet server was about 5 lines of C code. Granted, it's a sloppy patch, but this gives the Kismet server itself the ability to transport wireless IDS information via syslog.

Today, almost everything you put in your network has the ability to transport log information over a network. My personal belief is that it should be an option within Kismet itself. Even more so if Kismet is to be used as a serious wireless IDS system.

It's not that an application querying the Kismet API is a bad thing. Dragron pointed out that it could be done in just about any language (Perl, C, Ruby, Python, even bash). If I'd gone with the Kismet server/drone method, this might be acceptable. That's not the model I chose for reasons outlined above.

These will be field sensors. Even though this is only a PoC model, there is an assumption that these development utilities will be available. In the majority of cases for IDS/IPS systems, development tools are *not* likely to be present on these machines. At my place of employment, it's common *not to have Perl, Ruby, GCC (C compiler), Python, etc* loaded. That leaves me with one option. Bash, or shell scripting, to work with the Kismet API. That leaves a bad taste in my mouth, so I decided that would not be the way to go for me.

There is one other option. Everything we've done up to this point in our PoC can be supported by a package based system. For example, Snort, Barnyard2 and even Kismet updates or installs can be rolled out as packages. Hence, no development tools are needed on the "sensor" end of things.

I could write a custom C program that communicates with the Kismet API. I could then build that and roll it out as a package to the remote sensor. This idea I don't find so bad. Perhaps I'll come back to this some day. My patch seemed the most simple solution for my needs.

Kismet; Syslog me, baby

So far, we have almost all critical information of our wireless IDS system reporting back to a centralize MySQL database. We're missing one piece. That's the information Kismet provides about layer 2 attacks and other "networks" that "pop up" in our environment.

Again, our ultimate goal here is to build a unified console for wireless IDS. This means, that data being collected by Kismet's channel hopping/sniffing will ultimately need to be within the same database as our Snort information. From the database level, we already have 3

sensors running. They are:

- Sensor ID # 1 == Internet connected interface.
- Sensor ID # 2 == Internal wireless Access Point card monitoring internal use.
- Sensor ID # 3 == Kismet supplied TAP/TUN device. Snort is monitoring this interface that's being supplied by Kismet.

Our final sensor.....

- Sensor ID # 4 == Layer 2 data supplied from syslog via our patched version of Kismet.

At his point, we have not configured our syslog service to transmit data to our back end. However, there is more to our syslog data than Kismet information. That is, while Kismet will be supplying us information via syslog, so will other services. For example, OpenSSH session information gets logged via syslog. Firewall information (via iptables) might get logged via syslog. It would be best to have all that data.

So really, what Sensor ID # 4 will become is a 'syslog' interface. While most of it will probably be Kismet related, it'll also have other information we might find useful on our PoC wireless IDS system.

The first step to take is to make our PoC system send its syslog information to a centralized repository. As a log is generated, I want it sent to our back end. Not for correlation, but for archival purposes. If we do this correctly, even if an attacker somehow gains access to the system and alters the logs, we'll have a known 'good' copy. The logs are sent in 'real time', so even if the attacker mangles local logs, we'll be able to review something 'clean'.

At the time of this writing, the big players in the syslog field under *nix are syslog-ng and rsyslog. Below are links to example configuration files for each. Remember, "client" means our remote satellite wireless IDS system. This system will be sending the logs to the back end for collection. The "server" is the configuration for the side that will be receiving the logs.

Syslog-ng	**Client side**	**Server side**
Rsyslog	**Client side**	**Server side**

I'm using the the Rsyslog configurations. I've been a long time Syslog-ng user, but recently switched to Rsyslog due to some features it has over Syslog-ng. For example, in our Rsyslog client side configurations, I'm sending via the more reliable TCP method rather than UDP. This helps ensure the messages reach our back end servers. Rsyslog can also 'queue' events in the event our OpenVPN connection has died. On the Rsyslog server side, I'm storing events into a SQL database. This allows me to view *all* events from our wireless IDS sensor in a nice graphical interface. The interface I'm using is Loganalyzer. You could easily use something like Splunk or Logzilla to view syslog events.

That's getting a bit outside the scope of this document. While access to the archival syslog information is important, what I want are the threats to be located in one console. There is also plenty of documentation and 'HOWTOs' related to projects like Rsyslog and Loganalyzer.

Enter Sagan

One thing you might have noticed in our rsyslog/syslog-ng server/client examples, is a reference to 'sagan'. Sagan is ultimately the software that will 'patch' the information supplied from our Kismet server via syslog to our back end Snort based MySQL system. To make things a bit more clear, let me explain what Sagan is and what it does.

Sagan is a real-time correlation engine. What this means is that Sagan 'reads' logs, as it receives them, and attempts to correlate the information *within* your Snort SQL database with the packet/IDS/IPS data based on rules. For the sake of simplicity, lets examine what Sagan does.

Lets assume, for the sake of an example, you run an SMTP server. In front of this server, you use Snort as your IDS/IPS engine. So, as traffic is flowing to and from your SMTP server, that traffic also flows through your Snort sensor. For our example, lets assume a potential attacker is probing your network. They connect to your SMTP server, and issue the SMTP command "expn root". This traffic will be "seen" by Snort and "flagged" as an "attempted recon", and sent to your SQL back end. When Snort logs its information about the 'attempted-recon', it'll record information like, packet data (dump), source TCP/IP address, destination TCP/IP address, source TCP port, destination port, timestamp, etc.

Sagan, does the exact same thing, but instead of gathering the information about the 'attempted recon' at the network/packet level, it pulls the information from the log level. When the SMTP server sends out the log (via syslog), it'll typically have much of the same information. For example, the timestamp, TCP/IP destination and source, destination port (port 25) and sometimes even the source port!

Since Sagan has a lot of the same information as Snort, we can now *correlate log events with our IDS/IPS packet level events*! Sort of nifty, eh?

It gets better. Sagan can not only communicate directly with our SQL back end, but also store information into our Snort SQL database. Sagan uses a very similar rule set as Snort. Why is this important? *The same utilities you use for rule management with Snort work with Sagan.*. There's no additional software to load, only Sagan!

Note: If you're confused about what Sagan is and/or does, you might want to check out the Sagan: Log Correlation in a 'Snort like' way video at Securitytube.net. This is a short presentation that was given for the North Florida ISSA chapter in Jacksonville, Florida about Sagan.

Sagan configuration and installation

Configuration and installation of Sagan is pretty straightforward. However, if you need more detailed information, you'll want to check out Sagan's main site and the Sagan HOWTO.

Like Snort, we'll first need to compile and install Sagan. On our wireless PoC IDS system, we'll download the latest version of Sagan. This can be obtained from http://sagan.softwink.com/download/sagan-current.tar.gz.

Once download, we'll do the following:

```
$ tar -zxvpf sagan-current.tar.gz
$ cd sagan-0.1.7                    # Or whatever version is current
$ ./configure --disable-postgresql  # MySQL is enabled by default
```

```
$ make
$ sudo make install
$ sudo useradd sagan       # Used for privilege separation
$ sudo mkfifo /var/run/sagan.fifo    # Where to receive logs from
$ sudo chown sagan:sagan /var/run/sagan.fifo
$ sudo chown sagan:sagan /var/log/sagan
$ sudo chown sagan:sagan /var/run/sagan
```

Most of this should be straightforward. We want to add a user to the system "sagan", for similar reasons we add a 'snort' user. We'll be starting Sagan as 'root', but once Sagan is initialized, Sagan will 'drop' its privileges and 'become' the user 'sagan'.

This 'mkfifo' builds out FIFO (First In/First Out). This is also sometimes known as a 'named pipe'. This is where Sagan will be receiving log messages from either rsyslog or syslog-ng. Previously, in the syslog-ng and rsyslog "client" example configurations, the FIFO feature and Sagan templates were included. If you're confused, you might want to refer to them.

Note: Sagan is under constant development. If you run into a problem, report it. You might even want to check out the Sagan Git tree!

Next we'll want to pull down the latest Sagan rule set. The rule sets are very similar to Snort rule sets. They tell Sagan what to 'look' for in syslog messages. To get the latest rule set, simply do the following.

```
$ wget http://sagan.quadrantsec.com/rules/sagan-rules-current.tar.gz
$ tar -zxvpf sagan-rules-current.tar.gz
$ mv rules /usr/local/etc/sagan-rules
```

We can now configure Sagan. The key thing to look for is the database access ("output database:") plug in information. This tells Sagan how and what database (snort_beave) to access.

```
#  ,-._,-.     Sagan configuration file [http://sagan.quadrantsec.com]
#  \/)"(\/     By Champ Clark III & Quadrant InfoSec: http://www.quadrantsec.com
#   (_o_)      Copyright (C) 2009-2010 Quadrant InfoSec., et al.
#   /   \/)
#  (|| ||)
#   oo-oo


##############################################################################
# Standard _required_ Sagan options!
##############################################################################

# Sagan can read log entries via a FIFO (First In,  First Out) or via
# 'standard input' (stdin).   If this option is _present_,  then Sagan assumes
# log entries will come in via a FIFO.  If this option is _not_ present,  then
# Sagan will 'assume' log entries will come in via stdin.
#
# When Sagan takes input from stdin,  syslog-ng calls this via the 'program'
# mode/call.
#
# The --program flag can override this variable and force Sagan to use
# stdin.
#
# [optional] - Depending on system configuration.

var FIFO /var/run/sagan.fifo

# This variable contains the path of the Sagan rule sets.  It is required.

var RULE_PATH /usr/local/etc/sagan-rules

# Where Sagan should store it's lock file.

var LOCKFILE /var/run/sagan/sagan.pid

# This is for storage of Sagan runtime information.

var SAGANLOG /var/log/sagan/sagan.log

# Where Sagan should store alerts,  in a text file format.

var ALERTLOG /var/log/sagan/alert

# This is the IP address _of_ the Sagan system.   These options are used
# if Sagan is unable to determine a TCP/IP network address and/or port.
#
# [Required]

sagan_host 10.220.0.1
sagan_port 514

##############################################################################
# Snort database specific configurations
##############################################################################

# Sagan "sensor" configuration.  If you plan on running Sagan and storing data
# to a Snort database,  these options are required.   This information gets
# logged to the Snort database's "sensor" table to differentiate  between Snort
# IDS/IPS data and log data.   We don't really have an "interface", so we create
# one known as "syslog",  or whatever you'd like to call it.
#
```

```
# [Required if logging to a Snort database]

sagan_hostname wireless-sagan
sagan_interface syslog
sagan_filter none
sagan_detail 1

# If logging to a Snort database,  and the rule set specifies the protocol as
# "any",  this is the protocol we default to.  This is only needed if you are
# logging to a Snort database.
#
# Defaults to 17 [UDP],  which is what normal 'syslog' traffic is.  If you
# want TCP to be the desired effect,  change this option to "6".

; sagan_proto 17

# If you plan on logging to a Snort database,  this is where you tell Sagan
# where to log to.   The options should be pretty clear.  Currently Sagan
# supports MySQL and PostgreSQL.  The "maxdb_threads" is the maximum number of
# threads that can be created.  This defaults to 20.
#
# [Required if logging to a Snort database]

maxdb_threads 50
output database: log, mysql, user=sagan password=mypassword dbname=snort_beave host=1.1.2.70
; output database: log, postgresql, user=sagan password=secret dbname=snort_db host=192.168.0.1

##############################################################################
# External program/system calls configurations specifics
##############################################################################

# This option calls an external program when an event is triggered by a rule.
# Sagan basically makes a thread and calls the execl() system call.
# Data is supplied to the program being called via standard in (stdin).
# Data can be sent in a "parsable" or "alert" format.   "parsable" will be
# probably easier for your external program to parse.
#
# The "max_ext_threads" limits the amount of threads that can be created,
# which defaults to 20.

; max_ext_threads 50
; output external: /home/champ/stdout parsable

##############################################################################
# libesmtp (SMTP/E-mail) specific configuration options
##############################################################################

# If you'd like Sagan to e-mail events to you,  then you want to enable this
# option.  The fields should be pretty self explanatory.
# "max_email_threads" is the maximum number of threads that can be created
# for e-mailing out events.  This defaults to 20.  If min_email_priority is
# set,  then Sagan will _only_ e-mail event equal to or less than the
# minimum priority.   If left commented out or set to '0',  Sagan will
# e-mail all events.

; max_email_threads 50
; min_email_priority 5
; output email: to=sagan-alerts@example.com smtpserver=192.168.0.1:25 from=sagan@example.com

##############################################################################
# Logzilla (AKA : php-syslog-ng) database configuration specifics
##############################################################################

# If you'd like Sagan to log information to a Logzilla database (MySQL or
# PostgreSQL),  then you'll need to enabled this option.  The
# "max_logzilla_threads" option specifies the maximum number of threads that
# can be used to log to a Logzilla database.  This defaults to 20.  The
# majority of the options should be pretty clear,  but the options "full" and
# "alert" have very specific functions.   The "alert" option sends _only_
# Sagan triggered events to the Logzilla database.   If "full" is used,  then
# _all_ events,  triggered or not,  will be sent to the Logzilla database.

; max_logzilla_threads 50
; output logzilla: full, mysql, user=sagan password=secret dbname=syslog host=192.168.0.1
; output logzilla: alert, postgresql, user=sagan password=secret dbname=syslog host=192.168.0.1


##############################################################################
# Sagan rule sets! Arrgh Villains! Sagan neither takes nor gives mercy!
##############################################################################

# This should be enabled!  "classifications.config" allows correlation between
# a short name (for example,  "attempted-admin") and a priority level
# (for example, "1").  "reference.config" gives your various places to learn
# more information pertaining to an alert.
#

include $RULE_PATH/classification.config
include $RULE_PATH/reference.config
```

```
# These are the specific rule sets of events which are of interest and require
# notification.  Tailor these to your specific needs and check from time to
# time for new rule sets that might be of benefit.

include $RULE_PATH/arp.rules
include $RULE_PATH/attack.rules
include $RULE_PATH/bash.rules
include $RULE_PATH/bind.rules
include $RULE_PATH/kismet.rules
include $RULE_PATH/hostapd.rules
include $RULE_PATH/ntp.rules
include $RULE_PATH/openssh.rules
include $RULE_PATH/squid.rules
include $RULE_PATH/su.rules
include $RULE_PATH/syslog.rules
include $RULE_PATH/tcp.rules
```

If you're interested in enabling other output plug in's, it'd be best to refer to the Sagan web site.

Like Snort, Sagan rules are broken down protocols and/or programs. There are actually many more rules supplied via Sagan, but they aren't needed for this project. For example, Sagan rules ship with "proftpd.rules", but we won't be running a ProFTP server. If we enabled those rules, we would be watching for things that'll never happen on our wireless IDS system. This will simply use CPU ticks and memory. Lets break down the rule set we're using.

Rule set	Description
arp.rules	We'll be using 'arpalert' (http://www.arpalert.org) on this system to detect MiTM attacks, ARP spoofing, etc.
attack.rules	Very generic 'attack' rules (NOOP detection, etc)
bash.rules	We run 'bash' with bashlogger (USE=bashlogger) enabled. This means all commands on the system are sent to our centralize syslog server. This also means we can detect suspicious activity (ie : "HISTFILE=/dev/null") and alert us. This can't be altered by the user.
bind.rules	We run a chroot cache only Bind DNS server. This will let us know if something funky is being attempted with bind.
kismet.rules	Should be obvious, but this will detect information from our kismet_server!
ntp.rules	Network Time Protocol rules.
openssh.rules	Log authentication failures, success, etc to our wireless IDS system. This will also help us keep a record of access to the system.
squid.rules	We do a lot of transparent proxying at sites. This will take information (syslog) from squid and analyze it for us.
su.rules	This include 'su' and 'sudo'. This will let us know when bad or good su or sudo commands are issued.
syslog.rules	Generic syslog rules. Everything from 'Interface entered promiscuous mode' to 'system is out of memory!'
tcp.rules	Mostly Linux based rules. For example, "TCP Treason unclocked!" messages.
hostapd.rules	Hostapd rules watch for problems with our AP and possible malicious traffic reported by this software.

As you can see, we're using Sagan for a bit more than just Kismet traffic. We're also using it to keep an eye on other functions of the system.

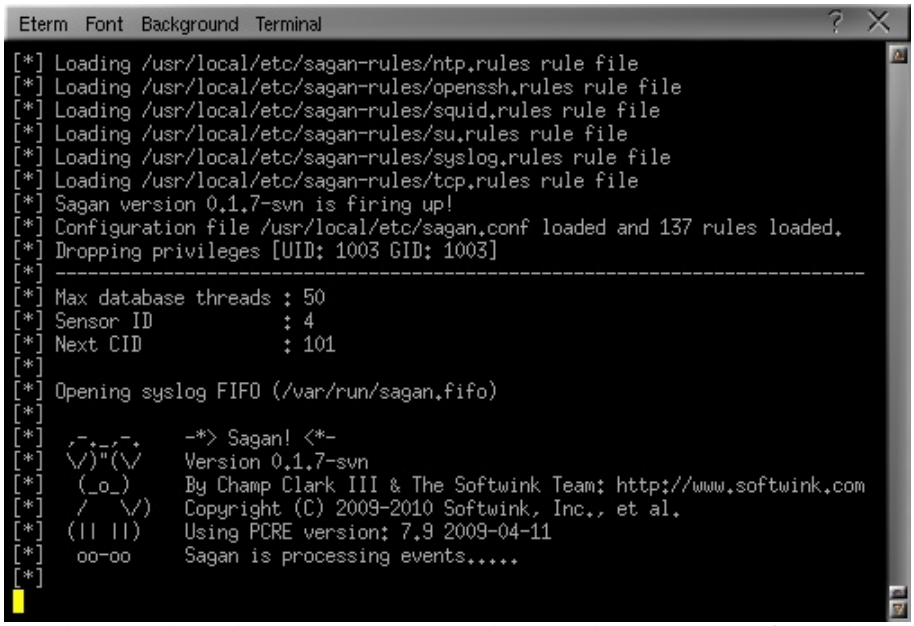
Once Sagan is installed and configured, we'll need to give it rights to the Snort database (snort_beave). Since Sagan works similarly to Snort, give it the same rights, just with another user name ("sagan"). On our back end server, we'll do the following. This will give our satellite wireless IDS sensors access to the database.

```
$ mysql -u root -p snort_beave
mysql> grant INSERT,SELECT on snort_beave.* to sagan@1.1.2.80 identified by 'mypassword';
mysql> grant INSERT, UPDATE, SELECT select on snort_beave.sensor to sagan@1.1.2.80 identified by 'mypassword'
mysql> exit
```

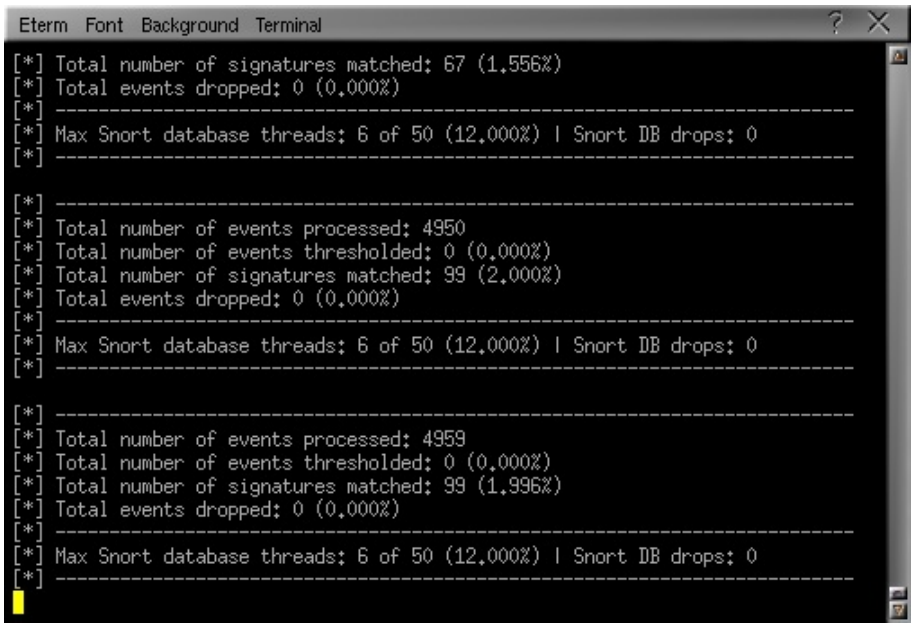
If everything is configured properly, we can now fire up Sagan and verify things are working. To do that, simply type (as "root"):

```
# /usr/local/bin/sagan
```

You should see something similar to the below:



(Figure 6. 'Sagan' startup splash screen. I'm using the SVN version of Sagan)



(Figure 7. Hitting a key in Sagan's interactive mode will dump statistics)

Note the "Total number of events processed: ". If that remains zero after some time of operation, Sagan probably isn't seeing the events correctly.

You can further check Sagan to see if it's operating properly by examining the Sagan 'alert' log. This is normally stored in the '/var/log/sagan/alert' file. If you're a Snort user, the output in the Sagan alert file should look pretty familiar, since Sagan uses a 'Snort like' alert syntax. Examining the 'alert' file, you should see something like:

```
[**] [5001014] [KISMET] Detected new managed network [**]
[Classification: suspicious-traffic] [Priority: 2]
2010-10-26 11:47:28 10.220.0.1:514 -> 10.220.0.1:514 user info
Message: Detected new managed network "2WIRE940", BSSID 00:25:00:00:00:00, encryption yes, channel 6, 54.00 mbit
[Xref => http://wiki.quadrantsec.com/bin/view/Main/5001014]
```

Once you're satisfied with Sagan's setup, you can tell Sagan to run in the background with the --daemon flag. To get statistics from Sagan, when running in the background, simply issue a 'killall -USR1 sagan'. Statistics will be stored in the /var/log/sagan/sagan.log file.

That's it! Final notes!

As a PoC system, we end up with a decent wireless IDS solution. Best yet, we're using nothing but open source software.

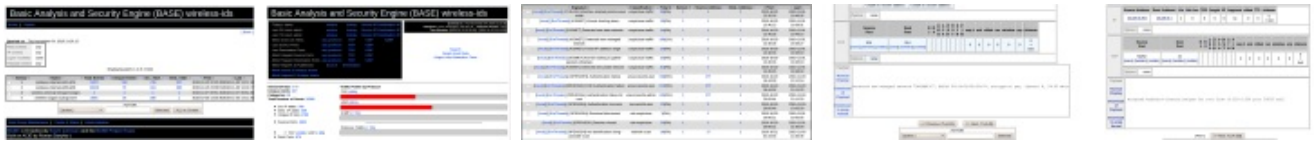
You've probably noticed that I've been dwelling on the fact that all potential attack or alert data must go to the same Snort MySQL database back end. Since we have all data in one place, we can now use the power of SQL to correlate that information. Another plus to this, we can use Snort web based front ends to examine the data. Yes, even the Kismet/syslog data!

With this mixture of software, we don't have to hunt through multiple consoles to do our job. We have *one, unified console from which to view potential threats.*

We can now take advantage of open source IDS consoles like BASE and Snorby. At Quadrant Information Security we use a proprietary console that queries the SQL database. Guess what. It doesn't matter. If your console, report generation utiltities, etc. can query a Snort database, they can be used with this mix! *It doesn't matter, it'll always work with this solution!*. Here's the short laundry list of things we've accomplished:

- We build a custom wireless access point with WPA2 enabled. This type of configuration using hostapd is extremely configurable. It should be simple to replicate and fit into just about any environment.
- Using Snort, we're monitoring all interfaces we directly use. We can monitor our internal wireless network for malicious traffic as well as our Internet connection for malicious traffic.
- We've actually taken the monitoring via Snort & Kismet to another level. We're monitoring traffic *outside* of our network for malicious traffic. While Kismet is useful for determining layer 2 attacks against out network, it's questionable if using the Kismet supplied TAP/TUN interface and Snort are necessary. In fact, during my tests, very little was seen on the Kismet TUN/TAP interface via Snort. This is probably overkill, but still made the project interesting. It might be best to leave Snort off the Kismet supplied TAP/TUN device and just let Kismet do the overall detection. Interesting, nevertheless.
- We're logging *all* syslog traffic to our back end. This is good from an audit stand point. If set up correctly, we can ensure that our logs are always "good", even if an attacker somehow gains access to our wireless IDS system.
- Sagan handles our monitoring of Kismet server output and relays it to our back end. Actually, Sagan does a bit more than that. He monitors not only syslog data from the Kismet server, but any syslog data that is seen.
- Since Snort and Sagan use a common 'type' of rule set, management becomes much easier. For example, for rule management, we can use the *same utilities (oinkmaster, pullpork) we use to manage Snort rules with Sagan rules!*
- One downside I encountered is that I wasn't able to control the transmit output levels very well from the wireless cards I chose. They work fine in my test environment and the range is good, but I doubt I'm able to put them up to full transmit power. Basically, pick your cards wisely.
- While Kismet is great for wireless detection, the IDS part of the system hasn't been updated in a bit. I brought this up to Dragorn, and he responded that attacks are either built off prior attack vectors or very difficult to detect. He's all for making the IDS portion of Kismet better. Also, to be fair, Kismet is the only active wireless IDS solution out there. During my tests, Kismet seemed to perform well in detecting layer 2 attacks. However, I didn't test very thoroughly. Maybe that'll be "part 2" of this article :)

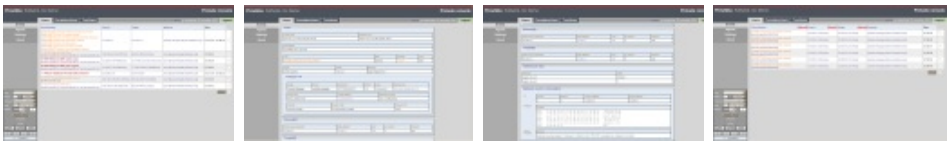
It wouldn't be fair to not show you the final outcome of this weekend PoC project. Below, I've included some screen shots of both BASE and Snorby working with our wireless IDS system with data supplied from both Sagan (syslog) and Snort (packet level).



BASE (Basic Analysis and Security Engine) Console

Snorby Console

Loganalyzer Console (Syslog data only)



Prelude's 'Prewikka' console. This is a new output plug in for Sagan and it based on the Prelude security frame work. Sagan & Prelude/IDMEF wasn't covered in this article but it's yet another console that is supported for our wireless IDS system.

About the author:

Champ Clark III (AKA - 'Da Beave') is a security researcher & engineer at Quadrant Information Security. He's the founder of the Deathrow OpenVMS public access cluster and a founding member of the VoIP hobbyist group Telephreak. He's an author of 'Asterisk Hacking' & 'Threat Analysis 2008' by Syngress Publishing. He regularly speaks at events like Defcon, HOPE, CCC (Germany) and various other events. He's the author of iWar, a VoIP/Unix based war dialer and Sagan, the log correlation and analysis engine talked about in this article. You can sometimes reach Champ Clark (Da Beave) on irc.2600.net / #sagan. You can also follow Champ via Twitter [http://twitter.com/dabeave666]

Thanks To

Bruce M. Wink and Kathrin Ritter at Quadrant Information Security for helping me get this article together. All the guys in the irc.freenode.net #kismet channel. In particular, Dragorn for the development of a great wireless scanner/sniffer. Also 'mmiller' and 'Zero_Cool' for input during the writing of this article. Of course, the irc.2600.net #telephreak crew.

Other articles by Champ Clark

Telco SMTP -> SMS/MMS Crypto (Telco related)
Installation & Securing VoIP with Linux (VoIP related)
Virtual Private Asterisk (VoIP related)

Videos by Champ Clark (Defon/HOPE/etc)

Hacking Internation Networks & Systems using VoIP (HOPE)
Sagan - Log correlation in a Snort like way (NF ISSA)
Automated Analog Telephone Logging (Defcon) PSTN based cartography (Defcon)