# FLoP - 1.6.0

## Fast Logging Project for Snort

**Dr. Dirk Geschke**

**Dirk@geschke-online.de**

**FLoP - 1.6.0: Fast Logging Project for Snort**
by Dr. Dirk Geschke

Published May 2006

# Table of Contents


**Abstract** ..... v

**1. Introduction** ..... 1

**2. Programs of the project** ..... 2

**3. The snort patch** ..... 4

3.1. Statistics with snort ..... 4

**4. Configuration of FLoP** ..... 7

4.1. Some notes on the configuration options ..... 7

**5. The programs sockserv and servsock** ..... 10

5.1. The details of **sockserv** ..... 10

5.1.1. Options ..... 11

5.1.2. Signalhandling ..... 13

5.1.3. Some additional notes ..... 13

5.2. The details of **servsock** ..... 14

5.2.1. Options ..... 15

5.2.2. The configuration file of **servsock** ..... 18

5.2.3. Signalhandling ..... 22

5.2.4. Some additional notes ..... 22

**6. The programs alert and drop** ..... 25

6.1. The details of **alert** ..... 25

6.2. The details of **drop** ..... 25

6.3. The command line options of **alert** and **drop** ..... 26

6.4. The configuration file for **alert** and **drop** ..... 27

6.5. Signalhandling ..... 29

**7. The program getpacket** ..... 31

7.1. The extension of the database scheme ..... 31

7.2. The command line options of **getpacket** ..... 32

7.3. The configuration file of **getpacket** ..... 33

7.4. Some final notes on **getpacket** ..... 34

**8. The program fpg, a *f*alse *p*ositive *g*enerator** ..... 35

8.1. The details of the **fpg** program ..... 35

8.2. The command line options of **fpg** ..... 36

8.3. Some final remarks on the program **fpg** ..... 37

**9. The `contrib` directory** ..... 39

9.1. The program **rules.pl** ..... 39

9.1.1. The options of **rules.pl** ..... 39

9.1.2. The configuration file `rules.pl.conf` ..... 40

9.2. The files `create_mysql` and `create_postgresql` ..... 41

9.3. The *cgi* files ..... 41

9.4. The perl script `stats.pl` ..... 42

**10. Summary of the tools and a final survey** ..... 43

# List of Examples

3-1. A simple perl script to feed an RRDtool database with a time step of 30 seconds. Here we only account for the rate of received packets but it is easily extended to use the other data. ....................5

# Abstract

The design of *snort* (http://www.snort.org/) requires a sequential work in the preprocessors, detection engine and output plugins for each network packet generating an alert. To enhance the detection capabilities of snort it would be an advantage to decouple the output plugins from the snort process. This is one feature of the *FLoP* project.

The second target regards the collection of alerts generated by several sensors on one *central server*. On this server all alerts are inserted into one *database* for further processing, analyzing and/or archiving. The processes buffer all alerts until they are spooled to the *central server* or are inserted in the *database*.

# Chapter 1. Introduction

The network intrusion detection system *snort* (http://www.snort.org/) captures network traffic. Each of those packets is first processed by the preprocessors. Here, among other things, the packets are reassembled on IP or TCP basis or are normalized like http traffic. After this stage the packet is either discarded (for the snort process) or forwarded to the detection engine. The detection engine applies several rule sets on this packet. If one rule matches an alert is generated and all output plugins are called sequentially to process this packet and the related informations like which rule generated the alert.

After the whole chain is worked through the next network packet can be analyzed. All packets arrived in between have to be buffered either by the kernel or the *libpcap*. If there are too many network packets and/or snort takes too long for processing the individual packets (or one output plugin blocks) it is likely that some packets are dropped.

So on a heavy network attack a lot of packets may be dropped due to the fact that snort is working on the output processing. On the other hand if there is no traffic snort will be idle.

One solution is to decouple the output plugins from snort. Why should snort bother about the various formats of alerts or how to insert the packets in a database? It would be of a great advantage to restrict snort to only detect alerts.

This is where *FLoP* starts. It decouples the output plugins from snort, gathers all alerts and sends them to a central server. At the server they where collected and inserted into a database for further processing. Additionally all alerts are buffered until they are processed (or where explicitly dropped by a configuration parameter if too many alerts are buffered).

# Chapter 2. Programs of the project

The project actually consists of six programs and one patch for snort:

## The patch and programs of *FLoP*

`snort-2.x.x_patch`

This patch adds an output plugin to write the alerts via an unix domain socket[1]

**sockserv**

This program generates the unix domain socket to which snort can write the alerts. The received alerts are buffered and transmitted to a central server running **servsock**.

**servsock**

On the *central server* all alerts from all remote sensors are collected and written to a *database*. Additionally alerts with high priority can be written to an unix domain socket where another program receives these alerts and send them via email to a list of predefined recipients.

**alert**

Alerts received via an unix domain socket are collected and send to a list of recipients.

**drop**

If too many alerts are buffered a memory shortage can arise. To avoid this a low and high water mark can be set. If more than high water alerts are in the buffer as many alerts are dropped to an unix domain socket until the low water mark is reached. This program collects these alerts and sends them via email to a list of recipients or prints them to *stdout* if sending of an email fails.

**getpacket**

There is a possibility to store additonal information about the captured network packets in the database. If these informations are available then this program can rebuild a *pcap* file consisting of the original captured network packet. This file can be used with programs like tcpdump or ethereal. To use this feature the database scheme has to be extended. See `README.database` for more informations on this topic.

**fpg**

This False-Positive-Generator takes a *snort* configuration file and creates for nearly each rule a network packet able to raise an alert. This program is useful for performance and stress tests of the whole chain starting with snort and ending at the database.

The next sections explain all these programs, how they work and how they can be configured.

# Notes

1. All used unix domain sockets are of type *datagram* to avoid blocking if one process creating the socket is not available.

# Chapter 3. The snort patch

This patch is needed to activate an output plugin which enables snort to write all alert information and the suspicious network packet to an unix datagram socket. To apply the patch you need only to change to the snort source directory and use the command:

**snort-2.x.x$ patch -p1 < /path/to/FLoP-1.6.0/patches/snort-2.x.x_patch**

After **configure** and **make** the **snort** program understands a new option in the `snort.conf` file:

**output alert_unixsock_db:** `/tmp/snort[, all|log|alert]`

The parameter of this output plugin describes where the unix domain socket should be found. Since we use unix domain sockets of type *datagram* it is not required that this socket exists. If there is no such socket, snort will simply write a warning message and continue to work. If the socket gets created in between, snort will use it. So snort is never blocked by this output plugin (except the reading process is explicit blocking).

Since snort-2.1.3 there exists also the possibility to write alternatively the `log` packets to the socket or both. If `all` is mentioned then only one packet is written to the socket if they are in both output chains.

Note: FLoP does not distinguish between *log* and *alert*. Therefore both kinds are ment if an *alert* is mentioned in this document!

Further there is the option `-Y` added to snort to avoid writing any alerts to the file system. (Before snort-2.3 this option was `-Q` but now `-Q` is used for the snort-inline part.)

Note: If you use the option `all` this is not necessary. If an output plugin is activated the *normal* ouput plugin is disabled. So if `all` is used then no alert or log data are written to the file system.

If you use `alert` then you can disable writing of log informations with the snort option `-N` or the equivalent `-K none`. If you only use the `log` then you can disable the alerts with the snort option `-A none`.

So this additional option `-Y` is no longer necessary but is still part of the snort patch.

The `log` facility is necessary if you want to store tagged packets or packets of a *dynaimc* rule in the database. Take also a look at the program **getpacket**.

## 3.1. Statistics with snort

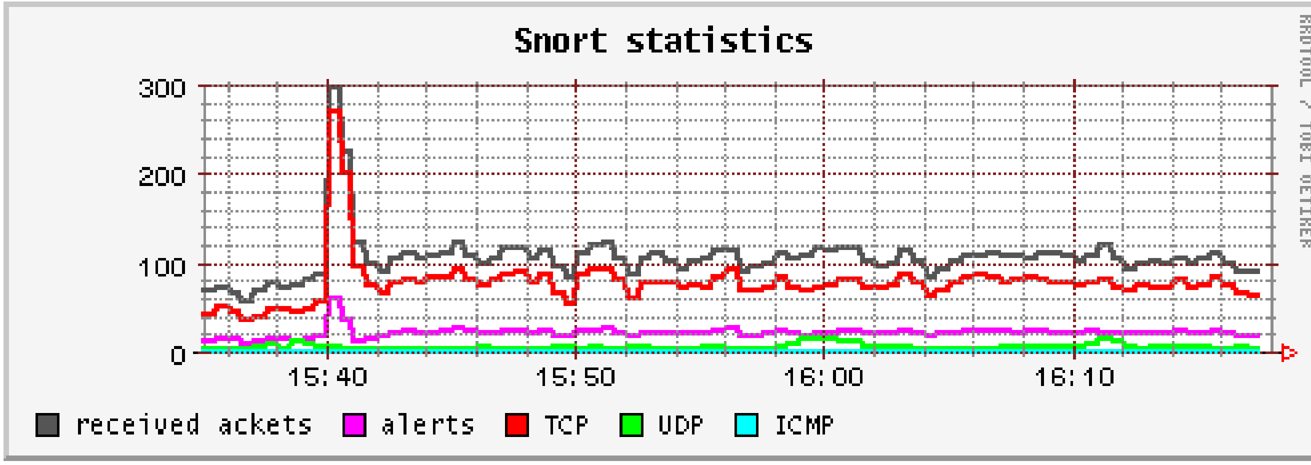
The patch additionally extends snort by a `-x` option (before snort-2.4 this option is `-Z`). This enables snort to write statistical inforamtion about the actual status to the unix domain socket `/tmp/stats`. These informations include the number of received and dropped packets, how many alerts where generated and which protocols where involved since the last time. The time intervall is the parameter after this option.

With the command

**snort -x 30**

the statistics are written every 30 seconds to the special unix datagram socket. Again, if this socket is not available, nothing will be written but snort will still work.

This information can be used in conjunction with the *RRDTool* (http://people.ee.ethz.ch/~oetiker/webtools/rrdtool/) to create some nice pictures like:




Statistics picture from snort generated with *RRDTool*

**Example 3-1. A simple perl script to feed an *RRDtool* database with a time step of 30 seconds. Here we only account for the rate of received packets but it is easily extended to use the other data.**

```
#!/usr/bin/perl
use IO::Socket;
use IO::Handle;
use Socket;
use RRDs;

$UXSOCKADDR="/tmp/stats";
```

```
unlink($UXSOCKADDR);
$sock = IO::Socket::UNIX->new( Local => $UXSOCKADDR, Type => SOCK_DGRAM) ❶
  or die "Can’t bind to Unix Socket: $!\n";
  $sock->setsockopt(SOL_SOCKET, SO_RCVBUF, 65440);          ❷
print "Ready to accept conntections!\n";

$RRDrecv="recv.RRD";

if (! -e $RRDrecv)                                          ❸
  {
     $CreateRRD=true;
  }
while (1) {
  $len=44;
  $sock->recv($input,$len);
  $TotalEvents++;

  @fields=unpack(" L L L L L L L L L L L",$input);
  print "\n";

  if ($CreateRRD eq true)
    {
      RRDs::create ("$RRDrecv", "--start", "$fields[0]", "--step", "30",❹
                    "DS:Statistics:GAUGE:61:0:U", "RRA:AVERAGE:0.5:1:100",
                    "RRA:AVERAGE:0.5:10:24", "RRA:AVERAGE:0.5:20:144");
       $CreateRRD=false;
     }

    RRDs::update ($RRDrecv, "$fields[0]:$fields[1]");       ❺
}
```

❶ Open an unix domain socket of type *datagram* to be able to receive data from snort.

❷ Increase the receive buffer of the socket.

❸ Test if a RRD database exist, if not we have to create one.

❹ There is no RRD database, so we create one here.

❺ Update the RRD database.

Note: These and maybe more informations could be received via the snort preprocessor *perfmonitor*. But this option is older than the preprocessor and therefore it is still part of the snort patch. Maybe this option should be removed in favour of *perfmonitor*.

# Chapter 4. Configuration of FLoP

After the snort sources are patched you have to run **configure** in the snort source directory. This will create the file `config.h` which is needed to compile FLoP. Both, snort and FLoP should use the same types of variables.

After this is done change to the FLoP directory and call here **configure**. You have to mention the path to the snort sources with the directive `--with-snort=`*/path/to/snort* and at least one database: Either Mysql (`--with-mysql=`*/path/to/mysql*) or PostgreSQL (`--with-postgres=`*/path/to/postgresql*).

Further you have to decide if the features and programs **drop** (`--enable-drop`, this is now default), **alert** (`--enable-alert`, this is now default), **getpacket** (`--enable-getpacket`) and **fpg** (`--enable-fpg`) should be compiled. To build **fpg** you must have libnet version 1.1 or newer.

## 4.1. Some notes on the configuration options

Whereas the path to the snort sources and the type of database are required some others are optional and some are recommended.

### The configure options in detail

`--prefix=`*`DIR`*

Gives the prefix to the installed binary, manual pages, documentation files and configuration files. These are stalled in *`DIR`*/bin, *`DIR`*/man, *`DIR`*/doc and *`DIR`*/conf.

`--with-snort=`*`DIR`*

This option is required. *`DIR`* should point to the configured snort sources. These are required to build the FLoP package. At least we need `config.h` of the snort sources. Additionally there is a little test to see if the patch is applied.

`--with-mysql=`*`DIR`*

This option activates the support for the *MySQL* database. *`DIR`* should point to the *MySQL* directory where the header and library files can be found. If first tries to run *`DIR`*/bin/**mysql_config** to get the compiler settings and flags.

`--with-postgres=`*`DIR`*

This option activates the support for the *PostgreSQL* database. *`DIR`* should point to the *PostgeSQL* directory where the header and library files can be found. The first try is to run *`DIR`*/bin/**pg_config** to get the compiler settings and flags. Note: You can activate both databases. You have to decide within `servsock.conf` which one should be used.

`--with-libbind`

This enables the use of libbind during the link process. Since the programs can use the library functions `getipnodebyname()` and `getipnodebyaddr()` which are not part of every operating system we can use this library for these functions. If this option is not activated then the functions `gethostbyname()` and `gethostbyaddr()` are used instead. So probably you will not activate this configure option.

`--enable-drop`

This enables the build of the program **drop** and activates the interfaces in **sockserv** and **servsock**. Note: You have still to activate this feature via the command line options or the configuration file. So it is save to enable this feature and therefore it is activated by default.

`--enable-alert`

This enables the build of the program **alert** and activates the interfaces in **sockserv** and **servsock**. Note: You still have to activate this feature via the command line options or the configuration file. So it is save to enable this feature and therefore it is enabled by default.

`--enable-getpacket`

This enables the build of the program **getpacket** which is able to rebuild a file with the network packet in *pcap* format from the database. Note: You have to extend the database scheme to use this feature and have to advise **servsock** to store the additional needed information in the database.

If you want to use *`libpcap`* to build the pcap file from the database you have to use the option `--with-libpcap`. But normally this is not necessary, **getpacket** is able to build the pcap file without the use of *`libpcap`*.

`--enable-fpg`

This enables the build of the program **fpg**. To compile this program you need the libnet library version 1.1 or newer. Since the API of libnet seems to change quiet frequently it is not unlekely that it will not compile clean. Therefore the option to compile this program is disabled by default.

`--enable-prepare`

This enables the use of the `PREPARE` command in conjunction with the *PostgreSQL* database. The effect is that all *SQL* statements where prepared, it is no longer necessary to do a type chech and similar operations. This may enhance the performance of the database access.

Note: This is not implemented for the *MySQL* database.

Note: It seems that at least with the actual snort database scheme this kind of optimization is already done by the client library of *PostgreSQL*.

`--enable-cache`

This enables the use of a *cache* for the `sig_ids`. If an alert is to be inserted the first thing is to ask for the sig_id of the signature. This is used for all further `INSERT`s. Since this is the only *read* statement it slows down the database access. The idea is to store all used `sig_ids` in a *cache* to avoid further `SELECT`s for them. This *cache* is implemented as a red-black binary tree.

`--with-maxclients=`*`clients`*

This option is specifies how many sensors can connect to **servsock** simultaneously. The default are 25 sensors or alternatively **sockserv** processes.

On some systems the database library and header files are already part of the operating system. There it can happen that for example the mysql header files are not found in `/path/to/mysql/include/`. Here you may find them in `/usr/include/mysql` where the compiler will not search for this headers by default. Thererfore it may be useful to set the `CPPFLAGS` together with the **configure** command:

```
CPPFLAGS=-I/usr/include/mysql ./configure --with-mysql=/usr ...
```

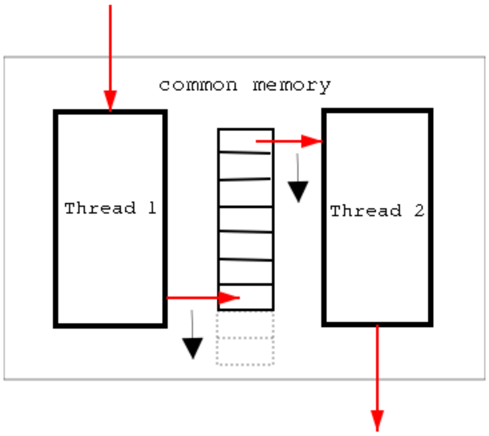
Additionally the options `CFLAGS` for compiler flags and `LDFLAGS` for linker options may be useful.

NOTE: If **mysql_config** or **pg_config** is used to get the compiler settings and flags you should not need to adjust these FLAGS.

For further information read the file `INSTALL` and the various `README` files of the distribution.

# Chapter 5. The programs sockserv and servsock

These two programs are very similar and work with two parallel threads. One thread receives the alerts and the other processes these data.




The principal of the **sockserv**/**servsock** process.

The first thread of the program **sockserv**[1] receives alerts from **snort** and stores them in a buffer in memory. The second thread takes these alerts and forwards them via *TCP/IP* to the **servsock**[2] program. This program consists of a master program waiting for connections from **sockserv** processes of remote sensors. For each connection one process is forked off. Each of these processes consist of two threads. One thread simply receives the incoming alerts, the second stores them to the database.

Starting with version 1.5.0 there is a third thread running, a control thread. This thread listens on a *named pipe* which can be used to change several options during runtime. See the file `README.ctrl` for further informations.

## 5.1. The details of sockserv

This program provides an unix domain socket for **snort**. One thread simply receives alerts via this socket and stores them in memory, see picture **sockserv**/**servsock** process.

Through the threading design and the use of a memory buffer the risk of loosing alerts is minimized. The output plugins from **snort** are reduced to a simple write statement on an unix domain socket. If more alerts are generated than **sockserv** can send to the *central server* these alerts are buffered in memory until the attack flood decreases.

To reduce the problem on memory shortage due to an high attack flood, the maximum number of alerts in the buffer can be limited. This is done via two parameters, the *`LowWater`* and *`HighWater`* marks. If more alerts than the *`HighWater`* mark are buffered in memory as many alerts are dropped until the *`LowWater`* mark is reached. All dropped alerts are written to an unix domain socket. The program **drop** is able to provide such a socket, receive these alerts and send them via email to a list of recipients. Otherwise, if you do not use **drop**, the these informations are written to *stdout* or *syslog*.

If either **sockserv** can not connect to **servsock** on startup or the connection is closed during runtime the program tries to reopen the connection after a short delay for several times.

Note: If the buffer of **sockserv** is empty at this point of time it tries a reconnect only if a new alert arrives.

All output can be redirected to *syslog*, using the facility `LOCAL0` and level `INFO`. A *daemon mode* is also supported. Finally statistics could be printed on a periodical basis or once by using the named pipe of the **sockserv** process.

## 5.1.1. Options

There are several options available:

```
sockserv [-bhlqv] [-A delay] [-D dropsocket] [-H HighWater] [-I interface]
         [-L LowWater] [-m mode] [-M maxtry] [-N sensorname] [-p port]
         [-P pidfile] [-s snortsocket] [-S server] [-V area,level] [-w dir] [-W waittime]
```

### The sockserv options in detail

`-A` *`delay`*

Print every *`delay`* seconds statistics about received, sent and dropped alerts. The change of these values between *`delay`* seconds is printed in brackets. See also option `-l`.

`-b`

Start the process in the background: *daemon mode*. This automatically activates option `-l`.

`-D dropsocket`

If there are more than *`HighWater`* alerts buffered then the newest alerts are dropped to `dropsocket` until the *`LowWater`* mark is reached.

`-h`

Print a help message and exit.

`-H` *`HighWater`*

Sets the *`HighWater`* mark, see option `-D`. The default value is 10000.

`-I` *`interface`*

Interface on which snort is sniffing. This parameter is optional but together with the sensorname, see Option `-N`, it should be unique. By default this option is not used.

`-l`

Log statitiscs to *syslog* instead of `stdout`. See also option `-A`.

`-q`

Enable quiet dropping, no dropped alerts were written to the `dropsocket`, see option `-D`.

`-L` *`LowWater`*

Sets the *`LowWater`* mark, see option `-D`. The default value is 9900.

`-m` *`mode`*

Sets the umask to *`mode`* for the daemon mode. This affects the mode for the created unix socket and PID file. The mode can be either given in *ascii*, *octal* (with leading `0`) or *hex* (with leading `0x`) format.

`-M` *`maxtry`*

Sets the maximum number of tries to (re-) connect to the *`server`*. See also option `-W`.

`-N` *`sensorname`*

Sensorname which should be used in the database. By default this is the hostname of the machine running **sockserv**. It is now possible to use more than one instance of **sockserv** per sensor. Note: You have to change the unix socket for different instances, see Option `-s`.

`-p` *`port`*

Defines on which *`port`* to try to reach the *`server`* running **servsock**. See also option `-S`.

`-P` *`pidfile`*

Filename to store the PID. Note: This file must be writeable by the user running **sockserv**!

`-s snortsocket`

Defines the name and directory where the unix domain socket is opened for snort. The default is `/tmp/snort`.

Maybe the default should be changed to something like `/tmp/snort_`*sensorname* if option `-N` is used.

`-S` *`server`*

Defines the server running **servsock**. The name can be either a full qualified domain name or an IP address. The default is `127.0.0.1`. See also option `-p`.

Note: The default is only useful for testing. In real productive systems you should use a separate server for the database and **servsock**.

`-V` *`area,value`*

Sets the debug level of the program. *`Area`* specifies the section of the code which should generate debug output. *`value`* should be between 0 (disabled) and 9 (maximum output). For further information read the file `README.debug`.

`-w` *`dir`*

Sets the working directory in daemon mode to *`dir`*. The default is to use the current working directory. It is useful to choose `/` to avoid blocking of mounted filesystems.

`-W` *`waittime`*

Time in seconds to wait between two tries to connect to the server. See also option `-M`.

## 5.1.2. Signalhandling

Currently the following signals are used with **sockserv**:

### Signals used with sockserv

`SIGINT`

Cancels the process, prints the final statistics and performs a clean exit. The *`socketname`* and *`pidfile`* are removed.

`SIGTERM`

This signal results in the same behaviour as `SIGINT`.

`SIGPIPE`

This signal is ignored. If the **servsock** program is interrupted during the data is sended. In this case we simply try to open a new connection and therefore we have to ignore this signal.

`SIGHUP`[3]

If this signal is received **sockserv** stops and restarts. First, if enabled, all buffered alerts are dropped via `dropsocket` and the final statistics are printed. Further *`socketname`* and *`pidfile`* are removed to enable a restart of the program. (Otherwise the program would fail since the id does not change!)

The actual version of **sockserv** uses a control thread to change some parameters during runtime. This thread is also used to print statistics on a periodically interval instead of using signals.

### 5.1.3. Some additional notes

The *drop* feature is not enabled by default and has to be compiled in separately. If it is not compiled in then the options `-D`, `-L` and `-H` are missing in the output of the `-h` option. It is highly advisable to choose a very large *`HighWater`* mark to buffer as many alerts as possible. This will reduce the possibility of information loss. On the other hand the difference between *`HighWater`* and *`LowWater`* should not be too large. To minimize information loss the alerts are spooled via **drop** to a mail server. Normally this server is either located on the *central server* or is reached via this server. If there are too many alerts spooled to **drop** the emails become unreadable long.

Problems should only arise if the connection to the **servsock** program is lost for a longer period. But if there are network problems then it is alike that **drop** will fail too. If this happens then the alerts are written either to *stdout* or *syslog*.

Be cautious: With increasing buffer usage the memory consumption raises with about 3 kB for each alert (actually 1048 bytes per alert on an *x86_32* System plus payload). But this memory is shared with the **snort** process. So set the *`HighWater`* to a value where it is safe for the snort process.

If a pid file exists then the program checks only for a running process with this PID. If one process is found the program exits. There is no check for which program is running, only if one with this PID exists!

## 5.2. The details of servsock

This program provides an TCP socket for **sockserv**. After a **sockserv** process has successfully connected a child process is forked off for this communication. The child process consists of two threads. One thread simply receives alerts via the TCP socket and stores them in memory, see picture **sockserv**/**servsock** process. The second thread feeds the stored alerts to a *database*.

To successfully connect there are a few things which must be fullfilled:

- If the endianess of the sensor and central server are different then a connection is permanently refused. This does not work.
- There is only one remote sensor with the same sensor name (see option `-N` and interface (see option `-I` of **sockserv**) allowed. If a second sensor with the same sensor name and interface tries to connect the access is denied.
- If there are still not yet processed data from the last connection between the remote sensor and the central server then the connection is as long refused as these data are not stored in the database
- If the database is not available if a sockserv process tries to connect then the connection is refused temporarily.

- If there is a swap file available, then the connection is temporarily halted until the data of the swap file is read into memory.

Through the threading design and the use of a memory buffer the risk of loosing alerts is minimized. If more alerts are available than **servsock** can insert into the *database*[4] the alerts are buffered in memory.

To reduce the problem on memory shortage due to a high overload, the number of alerts in the buffer can be limited. This is done as with **sockserv** via two parameters, the `LowWater` and `HighWater` marks. If more alerts than the `HighWater` mark are buffered in memory as many alerts are dropped until the `LowWater` mark is reached. All dropped alerts are written to an unix domain socket. The program **drop** is able to receive these alerts and send them via email to a list of recipients.

If either **sockserv** can not connect to **servsock** on startup or the connection is closed during runtime the program tries to reopen the connection after a short delay for several times.

All output can be redirected to *syslog*, using the facility `LOCAL0` and level `INFO`. A daemon mode is also supported. Finally, statistics could be printed on a periodical basis or once by sending a `SIGUSR1` to the **servsock** master process. This process will advice all child processes to print the statitistics.

## 5.2.1. Options

There are several options available:

```
servsock [-bCdefhlnqTuv] [-A delay] [-c config] [-D dropsocket]
         [-H HighWater] [-L LowWater] [-m mode] [-M priority]
         [-p port] [-P pidfile] [-s snortsocket] [-S server]
         [-U alertsocket] [-V area,level] [-w dir] [-W SwapDir] [-Z TimeZone]
```

### The servsock options in detail

`-A` *delay*

Print every *delay* seconds statistics about received, sent and dropped alerts. The change of these values between *delay* seconds is printed in brackets. See also option `-l`.

`-b`

Start the process in the background: *daemon mode*. This automatically activates option `-l`.

`-d`

Dump the actual configuration on startup. This is useful if both, a configuration file (see ) and command line options are used in combination and for debbuging purposes

`-C`

Specifies the use of a red-black tree to cache sig_id values of the database. This speeds up the `INSERT`s in the database but you have to enable it during the configuration run with the `--enable-cache`. See also the configure file option `DBCache`.

`-c config`

Specifies which configuration file should be used. The default is `servsock.conf`

`-D dropsocket`

If there are more than *`HighWater`* alerts buffered then the newest alerts are dropped to `dropsocket` until the *`LowWater`* mark is reached.

`-e`

Specifies the use of `PREPARE` statements with the database. This may speeds up the `INSERT`s in the database but you have to enable it during the configuration run with the `--enable-prepare`. See also the configure file option `DBPrepare`. This works only with the *PostgreSQL* database.

`-f`

Store additional information in the database so that a *pcap* file can be created with the program **getpacket**. Note: You need an extended database schema to use this option. See the file `README.payload` in the distribution.

`-h`

Print a help message and exit.

`-H` *`HighWater`*

Sets the *`HighWater`* mark, see option `-D`. There is no default value used.

`-l`

Log statitiscs to *syslog* instead of `stdout`. See also option `-A`.

`-L` *`LowWater`*

Sets the LowWater mark, see option `-D`. There is no default value used.

`-m` *`mode`*

Sets the umask to *`mode`* for the daemon mode. This affects the mode for the created unix socket and pid file. The mode can be either given in *ascii*, *octal* (with leading `0`) or *hex* (with leading `0x`) format.

`-M` *`priority`*

Sets the required priority for alerts to be written to `AlertSocket`. The program **alert** is able to read these alerts and send emails to a list of recipients.

`-n`

Do not resolve the full qualified names of the sensors, use the IP addresses instead. This will avoid conflicts with the *database* if on a new connection the DNS resolution fails or resolves to another name.

`-p` *`port`*

Defines on which *`port`* **servsock** should listen, see also option `-S`.

`-P` *`pidfile`*

Filename to store the PID. Note: This file must be writeable by the user running **servsock**!

`-q`

Enable quiet dropping, no dropped alerts were written to the `dropsocket`, see option `-D`.

`-r`

Store additional information in the database so that a *pcap* file including all tagged packets relating to an alert can be created with the program **getpacket**. Note: You need an extended database schema to use this option. See the file `README.payload` in the distribution.

`-s` `socketname`

Defines the name and directory where the unix domain socket of the *database* is opened. A value of *NULL* results in an internal `NULL` pointer, this is useful in combination with *PostgreSQL*. It is also possible to use a TCP socket via *hostname*:*port*.

`-S` *`server`*

Defines the interface where **servsock** should listen on. The name can be either a full qualified domain name[5] or an IP address. The default is `0.0.0.0` to bind on all available and configured interfaces. See also option `-p`.

`-T`

Enable trust modus for the *database*. If set, it is assumed that the alert description is already part of the database. If this is not the case, all these informations are inserted. So it is safe to enable this feature unless the transfer of alert message is disabled in **snort**. But this (removing of the alert message within snort) is a very *experimental* feature and is usually disabled by default. (But would save 256 Bytes on the wire!)

`-U` `alertsocket`

Specifies where the unix domain socket of the alert program can be found, see also `-M`.

`-u`

Disables the use of the `alertsocket`. This is useful if the alert is activated in the configuration file but there is no **alert** program running. So it is only useful for debugging.

`-v`

Print version information.

`-w` *`dir`*

Sets the working directory in *daemon mode* to *`dir`*. The default is to use the current working directory. It is useful to choose `/` to avoid blocking of mounted filesystems.

`-W` *`SwapDir`*

Sets the directory where the swap file `sensor_`*SensorName* is created and alerts are buffered if the database connection is lost.

`-V` *`area,value`*

Sets the debug level of the program. *`Area`* specifies the section of the code which should generate debug output. *`value`* should be between 0 (disabled) and 9 (maximum output). For further information read the file `README.debug`.

`-Z` *`TimeZone`*

Specifies which timezone should be used to store the time in the database (local timezone versus UTC). A *`ZimeZone`* of zero means to use the same timezone as localtime, any other value would result in the use of UTC. The default is to use the local timezone.

## 5.2.2. The configuration file of servsock

Additionally to the command line arguments there are some options which must be set via the configuration file. At least the *database* configuration has to be set in the configuration file[6].

The command line options have precedence above the settings in the configuartion file. If an option is mentioned on the command line this value is used regardless of the settings in the configuration file.

On the other hand all parameters of the command line can be set in the configuration file (except option `-u`). So the command line options are more suitable for quick tests.

The format of the file is simple, the first word is a keyword and the second is the value. They are separated by a colon (:) or equal sign (=). White spaces are allowed in any number.

The values can be put in single (’) or double (") quotes, all between is used as the value with one exception. This exception is the comment sign (#). All entries after this sign are ignored. To use the command sign it has to be escaped with a backslash: \#.

To use white spaces in a value they must be surrounded by quotes.

So all this results in a value with space, exept the last one without quotes. This will result in `spa`:

```
'spa ce' = "spa ce" = spa ce
```

All keywords are case insensitive (but not the values!).

## The parameters of the configuration file for servsock in detail

`DBuser`: *`name`*

Specifies the name of the *database* user who is allowed to do `INSERT`s, `SELECT`s and `UPDATE`s of tables. The default is *snort*.

`DBpassword`: *`password`*

Specifies the password used among with the `DBuser` name to connect to the *database*. Note: An empty password has to be represented by empty quotes, which is the default.

`DBname`: *`name`*

Name of the *database* where **servsock** should insert the alerts, defaults to *snort*.

`DBtype`: *`name`*

Type of the *database* to use. Actually only `MySQL` (http://www.mysql.com/) and `PostgreSQL` (http://www.postgresql.org/) are supported and have to be enabled at compile time of **servsock**. No default is set since it is not clear which *database* support was enabled at compile time of **servsock**.

`DBencoding`: *`name`*

Defines the encoding scheme wich is used to insert the payload into the *database*. Allowed values are `hex`, `base64` and `ascii`. The `base64` encoding requires less memory in the *database* but it makes it difficult to search for special entries in the payload. The `ascii`[7] encoding stores only ascii characters in the database, all binary data is replaced by a dot. So the only really useful option is the `hex` scheme which is the default or `base64` if saving of database space is desired.

`DBtrust`: *`value`*

A non-zero *`value`* enables the *trust* modus for the database. If this modus is enabled it is assumed that all possible signatures are already part of the database. This will result in slight faster `INSERT`s since less detailed `SELECT` statements are needed[8]. It is safe to enable this even if you are not sure, missing signatures will still be inserted. The equivalent command line is `-T`.

`DBCache`: *`value`*

A non-zero *`value`* enables the use of a *cache* for the *sig_id*. Since during inserts the only `SELECT`s are done to get the *sig_id*. The *cache* is implemented as a red-black tree. To use this option you have to enable it during the configure run with `--enable-cache`. The equivalent command line to activate the use of the *cache* is `-C`.

`DBPrepare`: *`value`*

A non-zero *`value`* enables the use of `PREPARE` with the database. This may speed up the `INSERT`s. To use this option you have to enable it during the configure run with `--enable-prepare`. This works only with the *PostgreSQL* database. The equivalent command line to activate this feature is `-e`.

`DBtrans`: *`value`*

A non-zero *`value`* enables the use of *transactions* together with the database. If you use the MySQL database you have to use tables of type *InnoDB*, otherwise the transactions are simply ignored.

`PIDFile`: `pidfile`

Specifies which file should be used to store the PID. This file must be writeable by the user running **servsock**! This correspond to option `-P`.

`SocketName`: `socketname`

This specifies where to find the unix domain socket of the database. If the word `NULL` (all capital!) is given, the database libraries find the socket by their own mechanism. This is useful in combination with the *PostgreSQL* database. This is equal to the `-s`. If `socketname` contains a colon (*:*) the first part is interpreted as a *hostname*, the second as a *port* number and a TCP connection to the database is used.

`ServerName`: *`name`*

Defines on which interface defined by the address **servsock** should listen on. Possible values for *`name`* are either full qualified names (not very useful) or a dotted IP address. The default is `0.0.0.0` to listen on all available interfaces.

`ServerPort`: *`value`*

Defines the port where **servsock** will listen on. The default is port `1234`. Compare to option `-p`.

`AlarmDelay`: *`value`*

Write every *`value`* seconds statistics of received, sent and dropped alerts. In braces the differences to the last output are printed. See option `-A`.

`Syslog`: *`value`*

If the *`value`* is non-zero then the statistics are logged via *syslog* and not printed to `stdout`. The facility is `LOCAL0` and the level is `INFO`. Compare to option `-l`

`FQNSensor`: *`value`*

With a *`value`* of zero the IP address of the sensor is used as sensor name in conjunction with the *database*. The equivalent command line option is `-n`.

`AlertSocket`: `alertsocket`

Name of the unix domain socket where alerts with high priority are written to. See option `-U`. If `alertsocket` has the name *NULL* then the alert feature is disabled.

`UnixPriority`: *`value`*

The value determines the minimum priority where alerts are additionally written to the `AlertSocket`[9]. The command line equivalent is the option `-M`.

`DropSocket`: `dropsocket`

Name of the unix domain socket where alerts are dropped to if the number of queued alerts reaches the `HighWater` mark. Compare to option `-H`. If `dropsocket` has the name *NULL* then the drop feature is disabled.

`DropQuiet`: *`value`*

If *`value`* is not zero then all dropped alerts are not written to the `DropSocket`. Note: Dropping is not disabled by this parameter.

`HighWater`: *`value`*

If the number of queued alerts reaches this *`value`* then **servsock** begins to drop alerts to the `DropSocket`. This corresponds to option `-H`.

`LowWater`: *`value`*

This *`value`* must be smaller than `HighWater`[10]. If the `HighWater` mark is reached so many alerts are dropped to the `DropSocket` until this `LowWater` value is reached. This corresponds to option `-L`.

`DaemonMode`: *`value`*

A non-zero *`value`* enables the *daemon mode*, the program forks into the background. This automatically activates the `Syslog` option. See option `-b`.

`Umask`: *`mode`*

Sets the *umask* to *`mode`* for the `DaemonMode`. This affects the mode for the created `PIDFile`. The *`mode`* can be either given in *ascii*, *octal* (with leading `0`) or *hex* (with leading `0x`). This is equal to the option `-m`.

`SwapDir`: `SwapDir`

Sets the directory where the swap file `sensor_`*SensorName* is created. This file is used to swap out alerts if the database has gone and is read in again if the database is available and the remote sensor connects again. The default is to use `/var/tmp`. See option `-W`.

`FullPayload`: *`value`*

Store additional information in the database so that a *pcap* file can be created with the program **getpacket**. Note: You need an extended database schema to use this option. See the file `README.payload` in the distribution and option `-f`.

`Reference`: *`value`*

Store additional information in the database so that a *pcap* file including all tagged packets can be created with the program **getpacket**. Note: You need an extended database schema to use this option. See the file `README.payload` in the distribution and option `-r`.

`Debug`: *`area,value`*

Sets the debug level of the program. *`Area`* specifies the section of the code which should generate debug output. *`value`* should be between 0 (disabled) and 9 (maximum output). For further information read the file `README.debug`. See option `-V`.

`TimeZone`: *`area,value`*

Specifies which timezone should be used. A value of zero for *`TimeZone`* results in the use of the timezone of localtime. Any other value will result in the use of UTC. See also option `-Z`.

## 5.2.3. Signalhandling

Currently the following signals are used with **servsock**:

`SIGUSR1`

Print statitics about received, sent and dropped alerts.

`SIGINT`

Cancels the master process, prints the final statistics and makes a clean exit. The *`socketname`* and *`PIDfile`* are removed. The child processes dump the buffered alerts to the swap file and exit.

`SIGTERM`

This signal results in the same behaviour as `SIGINT`.

`SIGHUP`[3]

If this signal is received by the master process then **servsock** stops each child process by sending a `SIGTERM` signal and restarts itself[11]. First all buffered alerts are written to the swap files and the final statistics are printed. Further `SocketName` and `PIDFile` are removed to enable a restart of the program. (Otherwise the program would fail since the PID did not change!) The child processes simply ignore the `SIGHUP` signal.

The actual version of **servsock** uses a control thread to change some parameters during runtime. This thread is also used to print statistics on a periodically interval instead of using signals.

## 5.2.4. Some additional notes

The *drop* and *alert* features are not enabled by default and have to be compiled in **servsock** separately. If it is not compiled in then the options `-D`, `-L` and `-H` are missing for the **drop** and the options `-M`, `-u` and `-U` are missing for the **alert** program in the output of the `-h` option.

In contrast to **sockserv** the `LowWater` and `HighWater` marks have to be choosen with more caution. First there are more processes running than the **servsock** processes especially the *database*. Further the bottleneck is not the network, it is usually the *database*. So it is quite normal that here the number of buffered alerts increase rapidly on heavy attacks.

Since the sensor name is taken from the IP address of the computer running **sockserv** (the remote sensor) there is only one **sockserv** instance per IP address allowed. Otherwise there will be a lot of collisions of inserts related to the *database*. (Two different sensors with the same name try to insert two different alerts with the same database Sensor ID, for example.)

If the connection dies, **sockserv** opens a new connection and a new **servsock** process is forked off. But if the old **servsock** thread feeding the *database* did not finished yet there arises a problem like the same sensor is logging twice times. Therefore **servsock** has a list of up to `25` (see the configure option `--with-maxclients` to adjust this value) running child processes with the sensor name they are dealing with. So if there is still one process running any new connection of a **sockserv** process with the same sensor name is rejected!

On startup a handshake must be fullfilled. During this phase the endianess of both partner, the availability of the database and the presence of a non-zero swap file are checked. Depending on the result a conncetion is either allowed, temporarily rejected or permanently denied.

If a `SIGHUP`[3] signal is received by the process with the PID stored in the `PIDFile` all child processes are terminated first. If there are buffered alerts it can take some time until all of them are written to the swap files. So a time delay on restart is not uncommon.

If either a `SIGINT` or `SIGTERM` is received by the **servsock** process handling a connection then all buffered alerts are written to the *swapfile*. Since this are threaded programs this may cause some problems with *Linux* using *linuxthreads* instead of the newer *NTPL* implementation. Note, the *NTPL* is only available with kernel *2.6*.

Nore: With the old *linuxthreads* you will see two processes for each thread, one in user land and one in kernel land instead of one process. Therefore it may cause some problems with the signal handler, not all PIDs behave the same.

If a `PIDFile` exists the program checks for a running process with the id of this file. If one is found the program exits to avoid running the same program twice. But there is no check for which program is running, only if there is one in the process list!

# Notes

1. This program povides an unix domain *sock*et and connects to a *serv*er.
2. This program povides an *serv*er and writes the alerts via an unix domain *sock*et to the database.
3. One important thing to obey is that either the program has to be started with absolute path or relative to the daemon working directory (option `-w`). Or the program has to be started without any path information at all and should be found in the system `PATH`. Otherwise the program will not find the own executable and will fail.
4. Sometimes databases hang on many inserts due to things like internal garbage collection. In addition there are many tables which have to be filled in for each alert. All this will slow down the insert rate of the *database*.

5. This not really useful since central server have usually more than on interface or you need a full qualified domain name for only this interface. Most name server resolve IP addresses in a round robin procedure for more than one IP address. So the interface on which **servsock** bounds would not be unambigous.
6. Especially the *password* for the database should not appear in the processlist.
7. This option should be removed in the future in favour of using `hex`, it is only available to be compatible with the database output plugin of snort.
8. This behaviour is a little bit different to the default one. Here we check or all values like revision and priority even if they are zero. In the other case we check for `NULL` values if they are zero. Indeed I think if the values are not set in the rule (aka the value is zero) this value should be inserted with the rule in opposite to keep it a `NULL` value. This behaviour was changed in FLoP-1.6.0.
9. This keyword should be replaced by `AlertPriority` in a future release.
10. The mimimum difference between this two marks should be at least greater than 10.
11. This results in a time delay for a restart since first it must be waited until all child processes exit.

# Chapter 6. The programs alert and drop

These two programs are very similar and are compiled out of the same source code. They provide an unix domain socket to receive alerts and try to send them via email to a list of defined recipients.

The alerts are buffered in memory before sending them via email. This can be triggered either on a periodically basis or if a given number of alerts is reached. Both variants can be activated separately but it is a good idea to use both. The time interval is useful to collect alerts instead of sending one separat mail for each alert which could result in a denial of service. The maximum number of alerts has the advantage to keep the used memory small and the emails in a readable size. Otherwise it could happen that too many alerts have to be stored in memory until an email could be send. So both options are useful in combination.

## 6.1. The details of alert

This program works in contrast to **drop** only with **servsock** and receives alerts via the unix domain sockets of priority equal or higher `UnixPriority`[1]. See also option `-M` of **servsock**.

The primary idea of this program is to have a separate mechanism to inform about critical alerts. Since it is very likely that the *database* is filled with a lot of less important alerts it is quite possible to either overlook the important alerts or to find them too late.

If the progam fails to send the emails it tries it again later. This is done up to five times. This number can be adjusted via the command line option `-M` or the `MaxCount` keyword.

If it is not possible to send an email within this time the program simply exits. Another process should inform an operator about this problem.

## 6.2. The details of drop

This program works in contrast to **alert** with both, **sockserv** and **servsock**. It receives alerts via the unix domain socket if the `HighWater` mark of queued alerts in `sockserv` or `servsock` are reached.

The primary idea of this program is to keep at least minimal informations about alerts. If there are too many alerts buffered some processes could fail due to memory shortage. So there should be a mechanism to drop some alerts to keep the buffer size limited. These alerts will not be inserted in the *database* but are mailed to a list of recipients.

If the progam fails to send the emails it tries it again later. This is done up to five times. This number can be adjusted via the command line option `-M` or the `MaxCount` keyword.

If it is not possible to send an email during this time the program writes the content of this email to `stdout`. Another process should inform an operator about this problem. In contrast to **alert** does this program not exit, it simply continues to work.

# 6.3. The command line options of alert and drop

Both programs use the same command line options, there is no difference between these options.

```
drop | alert [-bDFhlpTvV] [-A delay] [-c config ] [-d domain]
             [-f from] [-L level] [-m mode] [-M max] [-p port]
             [-P PIDfile] [-r rcpt] [-s socket] [-S server] [-w dir]
```

## The alert and drop options in detail

`-A` *`delay`*

Try every *`delay`* seconds to send an email if there are any alerts in the buffer.

`-b`

Start in daemon mode, switch to a background process. This automatically activates the option `-l`.

`-c config`

This defines the name of the configuration file to use.

`-d` *`domain`*

Use *`domain`* as `HELO` string on a connection to the MailServer, see option `-S`.

`-f` *`from`*

Sets the sender address of the emails to *`from`*.

`-F`

Try to resolve the sensor names via DNS.

`-h`

Print a help text and exit.

`-l`

Print via *syslog* instead of `stdout`.

`-L` *`level`*

If a number of *`level`* alerts are buffered then send an email. A value of zero disables this feature.

`-m` *`mode`*

Sets the umask to *`mode`* for the *daemon mode*. This affects the mode for the created unix socket and PID file. The mode can be either given in *ascii*, *octal* (with leading `0`) or *hex* (with leading `0x`) format.

`-M` *`maxcount`*

Specifies the maximum number of tries to send an email. If still no email could be send the program **alert** exits and the program **drop** prints all alerts to `stdout` or *syslog*, see option `-l`.

`-p` *`port`*

Try to reach the mail server on this *`port`*. The default is port 25, see also option `-S`.

`-P PIDFile`

Specifies which file should be used to store the PID. This file must be writeable by the user running **alert**/**drop**!

`-r` *`recipient`*

Sets the address of one recipient for the emails. This option can be used several times to build a list of recipients.

`-s socketname`

Specifies which unix domain socket of type *datagram* should be opened to listen for alerts.

`-S` *`server`*

Specifies the mail server which should be used to send the emails. This server should allow relaying for the server running **alert** or **drop**.

`-v`

Print version information and exit.

`-V`

Activates the verbose mode, some useful informations are printed if an email is sent. This is useful for debugging if there are any problems with the mail server.

`-w` *`dir`*

Sets the working directory in daemon mode to *`dir`*. The default is to use the current working directory. It is useful to choose `/` to avoid blocking of mounted filesystems.

## 6.4. The configuration file for alert and drop

The format of the configuration file is the same as for *servsock* and *sockserv*.

## The parameters of the configuration file for alert and drop in detail

`AlarmDelay`: *`time`*

The program will check every *`time`* seconds for the presence of received alerts. If there are any an email is send. The default is 5 minutes (300 seconds). The equivalent command line option is `-A`.

`AlarmLevel`: *`level`*

If the number of received alerts reaches *`level`* than an email is sent regardless of the status of `AlarmDelay`. The default is 0 which disables this feature. But it is recommed to use this feature since it limits the number of alerts which are buffered in memory. The command line option is `-L`.

`DaemonMode`: *`value`*

A non-zero value enables the daemon mode. The program forks off in the background and detaches from the terminal. See also option `DaemonDir` and `Umask`. This automatically enables also the option `Syslog`. The command line option `-b`.

`FQNNames`: *`value`*

A non-zero value enables resolving of full qualified names of the reporting sensor. To reduce CPU usage this values are cached in an internal list[2]. See also option `-F`.

`MailServer`: *`name`*

Specifies the server which should be used for relaying of the emails. This server should allow relaying for the different hosts running **sockserv** and **servsock**. The default server is `localhost`. The command line option is `-S`.

`MailPort`: *`number`*

Specifies that the mail server is reached via port *`number`*. The default is port `25`. The command line option is `-p`.

`MailRecipient`: *`address`*

Sets the address of one recipient of the emails. This option can be used several times to build a list of recipients. This is equal to the command line option `-r`.

`MailSender`: *`address`*

Sets the address of the sender of the emails. The command line option is `-f`.

`MailDomain`: *`domainname`*

Specifies the domain name which should be used in a mail session on startup (`HELO` string), see option `-d`.

`MaxCount`: *`count`*

Specifies the maximum number of tries to connect to the mailserver and deliver mails. After *`count`* tries the program **alert** terminates! The program **drop** simply writes all alerts to syslog or stdout and continues to work. See option `-M`.

`PIDFile`: `filename`

Specifies which file should be used to store the PID. This file must be writeable by the user running **servsock**! This correspond to option `-P`.

`SocketName`: `socket`

This specifies which unix domain socket should be opened for **sockserv** and **servsock**. This is equal to the `-s`.

`Syslog`: *`value`*

If the *`value`* is non-zero then all output is written to *syslog* and not printed to `stdout`. The facility is `LOCAL0` and the level is `INFO`. Compare to option `-l`

`Umask`: *`mode`*

Sets the *umask* to *`mode`* for the `DaemonMode`. This affects the mode for the created `PIDFile` and unix domain socket (see `SocketName`). The *`mode`* can be either given in *ascii*, *octal* (with leading `0`) or *hex* (with leading `0x`). This is equal to the option `-m`.

`DaemonDir`: `directory`

Sets the working directory in daemon mode to `daemondir`. The default is to use the current working directory. It is useful to choose `/` to avoid blocking of mounted filesystems. See option `-w`.

## 6.5. Signalhandling

Currently the following signals are used with **alert** and **drop**:

`SIGINT`

Cancels the program, the socket and PID file are removed and the program exits. The program **drop** prints all buffered alerts, either via `stdout` or *syslog*, see option `-l` or keyword `Syslog`, before it exits.

`SIGTERM`

This signal results in the same behaviour as `SIGINT`.

`SIGHUP`[3]

If this signal is received the unix domain socket will be closed, the socket and PID file removed and thre porgram gets restarted. The program **drop** prints first all buffered alerts.

`SIGALRM`

This signal is used to print statistics on a periodically basis. If this signal is send to the master process it is forwarded to all child processes.

# Notes

1. Be careful how you define the order of priorities. This has changed during the several versions of snort. So either 0 or 1 are the lowest or highest priority. You have to choose between these two variants!
2. If the DNS name changes while the program runs, the old names are still used. This is unlikely but the program may run for a long time.

# Chapter 7. The program getpacket

This program can build a network packet in *pcap* format which can be used by an analyzer like **tcpdump** or **ethereal**.

This requires some additional options to be used.

- The standard database scheme as shipped with snort must be extended.
- The payload has to be stored in *hex* format or in *base64*.The encoding in *ascii* is useless by design.
- The option `-f` of **servsock** or the parameter `FullPayload` in `servsock.conf` have to be enabled when the alert is stored in the database.
- Actually only *ethernet* is supported for the link layer. But to use another link layer is not really a problem. *Update:* The actual version support also the *linux cooked mode*.

The advantage of this approach is that the protocol analyzing mechanisms of programs like **ethereal** are far better than it is possible with *ACID/Base*. For example think of DNS queries or responses.

## 7.1. The extension of the database scheme

To store the additional header and pcap information in the database the normal scheme (as part of snort) must be extended. These extensions work well even with programs like *ACID/Base*.

These extensions must be done within the database, either with **mysql** or **psql**. If you have choosen the right database then enter at the command prompt the following commands:

**ALTER TABLE data ADD COLUMN data_header TEXT;**

This command adds a column for the missing packet headers. The payload stored by the normal process contains only the protocol payload of the alert. A *TCP* alert only stores the payload embedded in the *TCP* stream, no *TCP* header nor *IP* header nor the link level data.

**ALTER TABLE data ADD COLUMN pcap_header TEXT;**

This column stores the *pcap* header containing the time when the packet was captured and the snaplen.

**ALTER TABLE schema ADD COLUMN full_payload SMALLINT;**

With this column it is possible to note that the database is capable of storing the extended data.

**UPDATE schema SET full_payload=1;**

This sets the capability to store the full payload. If set to 1 then **servsock** will accept the `-f` option or `FullPayload` keyword.

Similarily, if the `-r` option or `Reference` keyword should be useable to store the reference of tagged packets then the event table has to be extended:

**ALTER TABLE event ADD COLUMN reference INT4;**

And the schema table has to be extended and updated so that we can query this settings.

**ALTER TABLE schema ADD COLUMN reference SMALLINT; UPDATE schema SET reference=1;**

If all this commands were applied to the database you have still to activate the storage of the additional data within **servsock**.

The `contrib/` directory contains two scripts which can build the database scheme 107 with all extensions. The created schemes are completely *ACID/Base* compatible: `create_mysql` and `create_postgresql`

# 7.2. The command line options of getpacket

`getpacket [-ahtvz] [-c ConfigFile] [-C` *`PacketCount`*`] [-S` *`SensorID`*`] [-w DumpFile]`

## The getpacket options in detail

`-a`

Build a pcap file of all packets with the same revision (tagged packets) which contain SID and CID. The option `-t` is automatically activated. Therefore you need an extended database scheme (see `README.payload`).

`-c ConfigFile`

Specifies which configuration file should be used. The default is `getpacket.conf` in the installation configuration directory. It is also possible to use the `servsock.conf` of **servsock**. The not needed keywords are ignored, only a warning is printed to `stdout`. This configuration file contains the data to needed to access the database.

`-C`*`CounterID`*

Specifies the counter *CID* of the alert in the database. Together with the sensor ID *SID* this data is unambiguous specified.

`-S`*`SensorID`*

Specifies the ID of the sensor *SID* in the database. Together with the *CID* is the data is unambiguous specified.

`-t`

Specifies that getpacket should attempt to use the reference column to include all the tagged packets relating to the initial *SID*/*CID* pair.

`-v`

Prints information about the version and exits.

`-w DumpFile`

Specifies which file is used to store the *pcap* data. If the special file name `"-"` is mentioned then the pcap data is written to `stdout`.

`-z`

Deactive the recreation of a pcap file with tagged packets. This way it is possible to disable the activation within the configuration file.

# 7.3. The configuration file of getpacket

## The getpacket keywords in detail

`DBuser`: *`name`*

Specifies the name of the *database* user who is allowed to do `SELECT`s of the tables. The default is *snort*.

`DBpassword`: *`password`*

Specifies the password used among with the `DBuser` name to connect to the *database*. Note: An empty password has to be represented by empty quotes, which is the default.

`DBname`: *`name`*

Name of the *database* where **getpacket** should select the alert packet data, defaults to *snort*.

`DBtype`: *`name`*

Type of the *database* to use. Actually only `MySQL` (http://www.mysql.com/) and `Postgres` (http://www.postgresql.org/) are supported and have to be enabled at compile time of **servsock**. No default is set since it is not clear which *database* support was enabled at compile time of **servsock**.

`SocketName`: `socketname`

This specifies where to find the unix domain socket of the database. If the word `NULL` (all capital!) is given, the database libraries find the socket by their own mechanism. This is useful in combination with the *PostgreSQL* database.

If the `servsock.conf` file is used then only the necessary keywords are used. All other options are ignored and a warning is printed to `stderr`.

## 7.4. Some final notes on getpacket

If the full payload is not stored in the database then an empty *pcap* file only consisting of a *pcap* file header is created. An error message is printed to **stderr**.

Some alert packets seem to have no payload (if you use ACID for example) but this is only for the higher level protocols valid. Only some preprocessor alerts have no payload at all since they may not act on a special network packet (e.g. the *sfPortscan* preprocessor).

The restriction to ethernet packets is only for the *pcap* header. Since the data link layer may have different sizes this must be entered in the *pcap* file header. But this information is not forwarded to the central server. This value can be easily adjusted. *Update:* The actual version support also the *linux cooked mode*.

Note: The rebuilt packet also contains the MAC addresses of the ethernet packet and the capture time of the host running **snort**.

If the reference data is not stored in the event table, getpacket cannot dump all related tagged packets in the pcap file.

# Chapter 8. The program fpg, a *f*alse *p*ositive *g*enerator

This program[1] creates network packets which raise false positive alerts within **snort**. It reads a **snort** configuration file and tries to build one network packet for each rule containing all necessary values.

Nearly all kind of network packets can be created, only some newer features of **snort** like `byte_test` and some `ICMP` types are not supported[2].

*Note:* There is an ongoing discussion about how useful or useless these kind of false positiv generators are. But at least there are some obviously reasons why they may be useful. For example, if you want to see what happens if the output plugins reach there limits you will need something which is able to generate enough alerts. Or think of finding problems in preprocessors or rules. If you find one this way then the use of such programs are more than justified. On the other hand, to use these kind of programs to estimate a limit up to where **snort** is able to perform his work may fool you since a lot of preprocessor works are ignored. So however, there are at least soem good reasons for these programs to exist.

## 8.1. The details of the fpg program

Actual fpg uses a lot of snort keywords. Up to 5 levels[3] of include files are supported.

**snort keywords used by fpg**

- include
- alert
- log
- var
- tcp
- udp
- icmp
- any
- rpc
- msg
- content
- uricontent
- dsize
- sameip
- offset

- distance
- depth
- within
- fragbits
- id
- ip_proto
- ttl
- itype
- icode
- icmp_id
- icmp_seq
- isdataat
- flags
- flow
- seq
- ack

Options not mentioned here are simply ignored[4]. You have explicitly to specify a source and destination address. So any special address in the configuration file are overwritten. So some rules will not raise alerts due to this wrong addresses.

# 8.2. The command line options of fpg

```
fpg [-hve] [-c config ] [-D count] [-n count] [-M maxpackets]
    [-R msec] [-T msec] -s source -d destination
```

## The fpg options in detail

`-c config`

Specifies which configuration file of **snort** should be used to generate the network packets. The default is `snort.conf` in the current directory.

`-d` *`destination`*

This option is mandatory and specifies the destination address used in the network packets. So any destination addresses in the configuration file are ignored.

`-D` *`count`*

Insert every *`count`* packets a time delay, see option `-T`. This feature is disabled by default.

`-e`

Run **fpg** in an endless loop, after the configuration file is worked through the program starts again at the beginning. The option `-M` is still valid. See also option `-n`.

`-h`

Print some help information and exit.

`-M` *`maxpackets`*

Specifies the maximum number of network packets to be generated and sent. See also `-e` and `-n`.

`-n` *`count`*

Send each build network packet *`count`* times. See also `-M` which is still valid and option `-e`.

`-R` *`msec`*

Specifies a random delay between two network packets of maximal *`msec`* milliseconds. This is useful to get a more random like traffic and to limit the rate.

`-s` *`source`*

This option is mandatory and specifies the source address used in the network packets. So any source addresses in the configuration file are ignored.

`-T` *`msec`*

Specifies the time delay between the number of network packets specified by the `-D` option. This is useful to avoid an overrun of the sending queue.

`-v`

Print version information and exit.

## 8.3. Some final remarks on the program fpg

Without any limitation and a fast machine the rate of generating network packets is much faster as the network device is able to generate. Therefore the options `-D` and `-T` were introduced [5].

The `-R` option was introduced to get a more realistic network traffic shape. This way it is possible to study the behaviour of **snort** on a more realistic scenario.

The `-n` option is the fastest way to generate a lot of alert packets, but all are equal. If one packet is build it is sent several times again. So all these packets look identical.

With the `-e` option the configuration file is walked through several times and all network packets are new build. Any unspecified values in the configuration file are replaced by random values. So with this option the network packets for the same rule look a bit different.

The destination address should be a valid one, there should exist a target with this address. Otherwise all packets will be blocked at the last hop with unsaturated arp requests for the destination address.

Be aware that nearly all packets will result in *Reset* or *ICMP* packets sent back to the mentioned source address (see option `-s`).

# Notes

1. To build network packets with own contents, e.g. different source addresses as the system has, TCP packets with flags set and so on, you must be root to use this progam!
2. To raise alerts within snort-2.0.0 you have to disable the `stream4` preprocssor. This preprocessor discards all packets which are not established and the rule says the packet has to be established.
3. This is only one parameter in the source file and can be easily increased.
4. These options are ignored, not the whole rule!
5. The *C* function `usleep()` is used, wich can sleep for microseconds. But the finest granularity of this function is in the range of 100 *Hz* (Starting with Linux-2.6 this value is adjustable up to 1000 *Hz* but some other operating systems may not be able to use other values). Therefore we use a delay in miliseconds every few packets instead of an `usleep()` after each packet is sent.

# Chapter 9. The `contrib` directory

This directory contains some more or less helpful programs and scripts for the various programs.

## 9.1. The program rules.pl

This is a very useful perl script which is able to fill the database with informations about all signatures, classifications and references.

The normal process is to ask the database if a signature is already stored in the database. This program can store all possible signature based alerts in the database. If this has been done and you are using option `DBTrust` with **servsock** then you can speed up the database access because you have only to ask if a *sig_id* to a given signature *id* (with the same priority and revision) exists, there is no need to ask for a given signature *message*.

If you additionally enables the use of a cache (`-C` or `DBCache` of **servsock** you can further increase the **INSERT** rate into the database.

The program first checks that the *classification* and *reference* are part of the database. The next step is to check if the signatures are already part of the database, if not then they were added. If not, then the new signature is added.

*Note:* Normally the priority of a signature is only estimated via the classification. Since the priority can be given individually and may have other values on different sensors the priority range can be adjusted.

The file `classification.pl` is a perl script which is more or less a framework which only updates the classification table in the database.

### 9.1.1. The options of rules.pl

There are only a few options available:

```
rules.pl [-h] [-c config] [-C classification.config ] [-P priorityrange]
         [-R reference.config] file1.rule file2.rule ...
```

Some value can be set via the configuration file mentioned by option `-c`, the default is `rules.pl.conf` in the actual directory. The files `classificaion.config` and `reference.config` are part of the *snort* distribution.

If the `-P` is used then all rules are inserted starting with priority one up to the given value.

Finally a list of files can be mentioned, they should all contain signatures in the form **snort** uses them.

## 9.1.2. The configuration file `rules.pl.conf`

There are only a few *keywords* which can be used by **rules.pl**, most of them regard the database access.

`$dbtype=`*`database`*;

Estimates which type of database should be used, this value can be either *mysql* or *postgres*.

`$dbname=`*`name`*;

Estimates which database should be used to insert the rules in.

`$dbuser=`*`username`*;

Connect as *`username`* to the database. This user must be able to **INSERT** into the database.

`$dbpass=`*`password`*;

This sets the password which is used with the *`username`* to access the database.

`$dbhost=`*`host`*;

This defines on which *`host`* the database is running, this may be empty. If no name is given then *localhost* is used.

`$dbport=`*`port`*;

This defines the port which should be used to address the database running on *`host`*. This may be empty in which case the default for the `DBType` is used.

`$ClassFile=`*`filename`*;

This defines the name and location of the `classification.config`. This is normally located in the `etc/` of the *snort* source distribution.

`$ReferenceFile=`*`filename`*;

This defines the name and location of the `reference.config`. This is normally located in the `etc/` of the *snort* source distribution.

`$PrioRange=`*`value`*;

If *`value`* is not set or zero then the normal priority as defined by the rule is used. Otherwise the whole range from 1 to *`value`* is stored in the database.

*Note*: The last entry in the `rules.pl.conf` must be *1;* since this file is included via **do "$conffile";**

An example file `rules.pl.conf` is located in the `contrib/` directory.

## 9.2. The files `create_mysql` and `create_postgresql`

These two files are similar to that distributed with *snort*. The main difference is that these are extended to store the full payload and reference id in the database. The refernce id is useful to recreate a traffic stream if they were save via a *tag* keyword in the rule.

A further enhancement is the support of scheme 107 which is able to save the *generator ID* so it can be clearly identified which part of **snort** raised this alert.

This files can be *sourced* from the command line interface of the database (**mysql** or **psql**). Usually the option `\i /path/to/create/file` is the relevant part.

*Note*: You still have to create a user and grant them the right privileges so that **servsock** can use the database.

## 9.3. The *cgi* files

The files `sum.pl`, `db-cgi.pl`, `ip.pl`, `list.pl`, `select.pl`, `signature.pl` and `dbh.pm` can be used to build a small web interface in order to query the database for some statistics and alerts. All you have to do is to put them in a cgi directory of your database.

All perl script expect that they are invoked via the directory `/cgi.bin/`. Further at the beginning of each script there is a `BEGIN` statement which extends the perl search paht. This path should contain the file `dbh.pm`.

The file `dbh.pm` defines the database access method. It should be easy to change the parameter in this file to the needed ones.

`sum.pl`

: This script gives a summary overview of the sensors in the database and the different kinds of alerts. This is useful as a starting point.

`db-cgi.pl`

: This is used to display on single alert.

`list.pl`

: This is used to display a list of search results. From each result the script `db-cgi.pl` can be invoked for displaying further details.

`ip.pl`

: This script is used to display results based on an IP address or protocol.

`select.pl`

This program can be used to fetch an alert from the database by a given sensor id *SID* and counter id *CID*.

`signature.pl`

This script fetches informations based on a special signature.

All these scripts are at least a framework and can be used as starting point to build some useful applications. So one idea was to create *HTML* links to an alert in the database which can be send via the **alert**. This can easily be implemented since the *SID* and *CID* are already known.

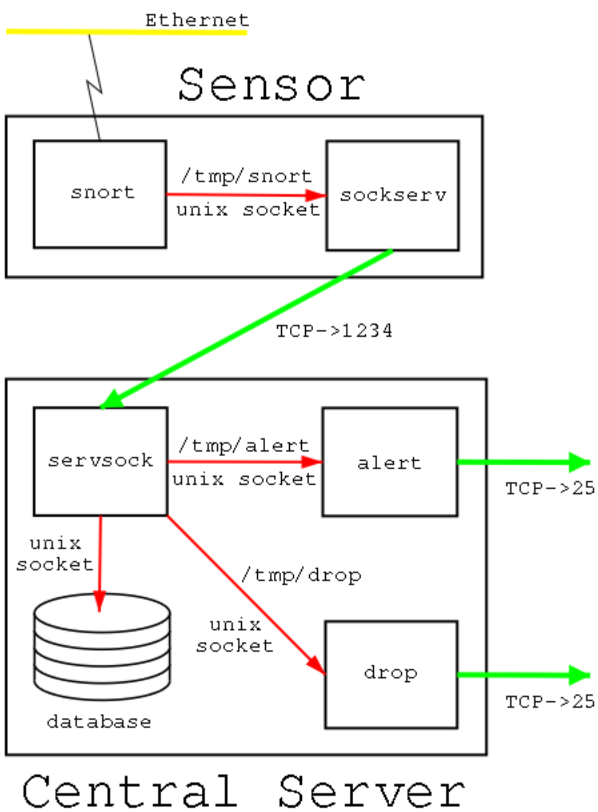
## 9.4. The perl script `stats.pl`

This is an easy script which can be used to feed an *RRD* file. The statistics were created with a similar script like this program.

But probably this is no longer needed or useful since *perfmonitor* preprocessor is available. But it is still part of this project.

# Chapter 10. Summary of the tools and a final survey

The picture shows how all these tools work together. **snort** watches the Ethernet wire for suspicious traffic and reports alerts to **sockserv** which forwards them to **servsock**. This program writes the alerts together with the payload in a *database*.




An illustration how **sockserv**, **servsock**, **alert** and **drop** work together [1].

The program **fpg** can be used to generate traffic on the ethernet which should raise alerts within **snort**. These alerts are written to the unix domain socket `/tmp/snort` where **sockserv** reads them.

One thread of **sockserv** reads in these alerts whereas the second thread sends the alerts via TCP (port 1234) to the *central sever*. All alerts are buffered to account for bottlenecks in the chain.

On the *central sever* the master process of **servsock** waits for new incoming connections from remote *sensors*. If a new conncetion is established a process is forked off to handle this commmunication.

One thread is of this process receives the alerts and stores them in a memory buffer. The second thread takes these alerts out of the buffer and stores them via an unix domain socket in the *database*. On alerts with a high priority the details and ID of this event are written to the unix domain socket `/tmp/alert`.

The program **alert** reads this alert informations and collects them. On a periodically basis or if a given number of alerts is reached this information is send via email to a list of recipients.

If there are too many buffered alerts within **servsock** a drop functionality is activated. If the `HighWater` mark is reached then as many alerts are written and dropped as many to `/tmp/drop` until the `LowWater` mark is reached.

The program **drop** reads these alerts and collect them. It works like **alert** but does not store the database ID since these alerts are not part and will not be part of the *database*. If the sending of mail fails for several times these alerts are written to `stdout` or *syslog* so no alerts should be lost. This behaviour is different to **alert** which would simply delete these alerts [2].

# Notes

1. The program **drop** can also work with **sockserv** but this is omitted in this picture.
2. The reason for this behaviour is quite simple: The program **alert** is intended to inform about alerts with high priority if they arrive. But these alerts are already part of the database. So if the sending of mail fails one can still find these alerts in the database.