



Software Engineering Institute | Carnegie Mellon

# Analysis Pipeline Handbook

Version 5.3.1

CERT Network Situational Awareness Group  
© 2010–2016 Carnegie Mellon University

January 22, 2016

Use of the Analysis Pipeline and related source code is subject to the terms of the following licenses:

GNU Public License (GPL) Rights pursuant to Version 2, June 1991  
Government Purpose License Rights (GPLR) pursuant to DFARS 252.227.7013

#### NO WARRANTY

ANY INFORMATION, MATERIALS, SERVICES, INTELLECTUAL PROPERTY OR OTHER PROPERTY OR RIGHTS GRANTED OR PROVIDED BY CARNEGIE MELLON UNIVERSITY PURSUANT TO THIS LICENSE (HEREINAFTER THE "DELIVERABLES") ARE ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, INFORMATIONAL CONTENT, NONINFRINGEMENT, OR ERROR-FREE OPERATION. CARNEGIE MELLON UNIVERSITY SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS LOSS OF PROFITS OR INABILITY TO USE SAID INTELLECTUAL PROPERTY, UNDER THIS LICENSE, REGARDLESS OF WHETHER SUCH PARTY WAS AWARE OF THE POSSIBILITY OF SUCH DAMAGES. LICENSEE AGREES THAT IT WILL NOT MAKE ANY WARRANTY ON BEHALF OF CARNEGIE MELLON UNIVERSITY, EXPRESS OR IMPLIED, TO ANY PERSON CONCERNING THE APPLICATION OF OR THE RESULTS TO BE OBTAINED WITH THE DELIVERABLES UNDER THIS LICENSE.

Licensee hereby agrees to defend, indemnify, and hold harmless Carnegie Mellon University, its trustees, officers, employees, and agents from all claims or demands made against them (and any related losses, expenses, or attorney's fees) arising out of, or relating to Licensee's and/or its sub licensees' negligent use or willful misuse of or negligent conduct or willful misconduct regarding the Software, facilities, or other rights or assistance granted by Carnegie Mellon University under this License, including, but not limited to, any claims of product liability, personal injury, death, damage to property, or violation of any laws or regulations.

Carnegie Mellon University Software Engineering Institute authored documents are sponsored by the U.S. Department of Defense under Contract FA8721-05-C-0003. Carnegie Mellon University retains copyrights in all material produced under this contract. The U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce these documents, or allow others to do so, for U.S. Government purposes only pursuant to the copyright license under the contract clause at 252.227.7013.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Using Schemas . . . . .	8
1.2	Configuration File Organization . . . . .	8
1.3	Configuration Order . . . . .	9
1.4	Configuration Syntax and Underscores . . . . .	9
1.5	A note on time values . . . . .	10
1.6	A note on this versions beyond version 3 . . . . .	10
<b>2</b>	<b>Installation and Integration</b>	<b>11</b>
2.1	Building pipeline . . . . .	11
2.1.1	Compiling and installing . . . . .	12
2.2	Data Source Configuration . . . . .	13
2.2.1	SiLK Records . . . . .	13
2.2.2	YAF Records . . . . .	14
2.2.3	Raw IPFIX Records . . . . .	14
2.2.4	Data Location . . . . .	14
2.2.5	Timing and Flow Grouping Options . . . . .	15
2.2.6	Data Source Configuration File . . . . .	15
2.3	Stats Log Updates . . . . .	16
2.4	Preparing to run . . . . .	16
2.4.1	Alerting with libsnarf . . . . .	18
2.4.2	Legacy Alerting . . . . .	18
2.5	Integrating with SiLK packing . . . . .	19
2.5.1	Using rwsender . . . . .	19
2.5.2	Using rwreceiver . . . . .	19
2.5.3	Using rwflowappend . . . . .	20
2.5.4	Using rwflowpack only . . . . .	20
2.6	Automating the Analysis Pipeline . . . . .	21
<b>3</b>	<b>Configuration Language</b>	<b>23</b>
3.1	Fields and Field Lists . . . . .	23
3.1.1	Fields with Multiple Values . . . . .	24
3.1.2	SiLK Records . . . . .	24
3.1.3	YAF Records . . . . .	26
3.1.4	IPFIX Records . . . . .	27
3.1.5	PMAPs . . . . .	27
3.1.6	Field Booleans . . . . .	28
3.1.7	DNS Derived Fields . . . . .	29
3.1.8	Timestamp Derived Fields . . . . .	29
3.1.9	Other Derived Fields . . . . .	30
3.2	Filters . . . . .	30
3.2.1	Operators and Compare Values . . . . .	30

3.2.2	Schema Specification for Filters . . . . .	32
3.2.3	Filter Examples . . . . .	32
3.3	Internal Filters and Named Lists . . . . .	33
3.3.1	Internal Filter Description . . . . .	33
3.3.2	Internal Filter Syntax . . . . .	33
3.4	Introduction to Evaluations and Statistics . . . . .	34
3.4.1	Id . . . . .	34
3.4.2	Schema Specification for Evaluations . . . . .	35
3.4.3	Alert Type . . . . .	35
3.4.4	Severity . . . . .	36
3.4.5	Filter Id . . . . .	36
3.4.6	“Binning” by distinct field: FOREACH . . . . .	36
3.4.7	Active status . . . . .	37
3.4.8	General Evaluation and Statistic Layouts . . . . .	37
3.5	Primitives . . . . .	37
3.5.1	Time Window . . . . .	38
3.5.2	Record Count . . . . .	39
3.5.3	Sum . . . . .	39
3.5.4	Average . . . . .	40
3.5.5	Distinct . . . . .	41
3.5.6	Proportion . . . . .	41
3.5.7	Everything Passes . . . . .	42
3.5.8	Beacon . . . . .	42
3.5.9	Ratio . . . . .	43
3.5.10	Iterative Comparison . . . . .	44
3.5.11	High Port Check . . . . .	44
3.5.12	High Port Check in a Statistic . . . . .	45
3.5.13	Web Redirection . . . . .	45
3.5.14	Sensor Outage . . . . .	45
3.5.15	Difference Distribution . . . . .	46
3.5.16	Fast Flux . . . . .	47
3.6	Evaluation Specific Detail . . . . .	48
3.6.1	Checks . . . . .	49
3.6.2	Outputs . . . . .	49
3.6.3	Alerting settings . . . . .	51
3.7	Statistic Specific Detail . . . . .	52
3.8	List Configuration . . . . .	53
3.8.1	Alert Triggers . . . . .	54
3.8.2	Output Files . . . . .	55
3.8.3	Other Options . . . . .	56
3.9	List Bundles . . . . .	56
3.9.1	Named lists for bundle . . . . .	56
3.9.2	Add element to another list . . . . .	56
3.9.3	Severity . . . . .	57
3.9.4	Do Not Alert . . . . .	57
3.9.5	List Bundle example . . . . .	57
<b>4</b>	<b>Example Configurations</b> . . . . .	<b>59</b>
4.1	Simple filters and evaluations . . . . .	59
4.2	Statistics . . . . .	61
4.3	Watchlists . . . . .	62
4.3.1	Alternate configuration . . . . .	64
4.3.2	DNS Watchlist . . . . .	65
4.4	Passive FTP detection . . . . .	66

4.5	Web server detection . . . . .	67
4.6	IPv6 tunneling detection . . . . .	68
4.6.1	Teredo . . . . .	69
4.6.2	6to4 . . . . .	70
4.6.3	ISATAP . . . . .	71
4.7	Chaining Lists . . . . .	71
<b>A</b>	<b>Manual Page</b>	<b>75</b>
A.1	NAME . . . . .	75
A.2	SYNOPSIS . . . . .	75
A.3	DESCRIPTION . . . . .	77
A.4	OPTIONS . . . . .	80
A.5	ENVIRONMENT . . . . .	85
A.6	SEE ALSO . . . . .	85



# Chapter 1

## Introduction

The Analysis Pipeline was developed to support inspection of flow records as they are created. The Analysis Pipeline supports many analyses, including:

- Watch lists (did we see traffic from a known bad IP?) (Section: 4.3)
- DNS query inspection (did a known malicious link get clicked?) (Section: 4.3.2)
- Network profiling (Section: 4.5)
- Beacon detection (Section: 3.5.8)
- Passive FTP detection (Section: 4.4)
- IPv6 tunnel detection (Section: 4.6)
- Thresholding (e.g., is total bytes over a limit?) (Section: 4.1)
- Collection issues (is a sensor no longer reporting?) (Section: 3.5.14)
- Fast Flux (requires YAF or IPFIX data with DPI info)

SiLK IPv4 records were the focus of versions 4.x and below. Version 5.0 expanded the data options to include SiLK IPv6 records as well as IPv4. Version 5.1 opens the door to a wider array of record types. The first specific records are those exported by YAF. YAF exports the core flow information used by SiLK, and enhances it with deep packet inspection information. Analysis Pipeline can also accept raw IPFIX records from any application. Any field used by YAF, or present in the IPFIX records, can be used by Pipeline. It can handle multiple sources, and multiple record types transmitted by each data source. If multiple sources or record types share a field or fields, Pipeline is able to recognize that and combine state for values of that field from each data source. If watchlisting on SIP for example, any data record that has SIP can meet the criteria for that filter.

The new data formats add an additional way to read data. As with earlier versions, it can accept a list of files on the command line and poll a directory for incoming files. It can also accept a socket connection for YAF and IPFIX records. If a socket or directory polling is used, Pipeline will run as a daemon unless `--do-not-daemonize` is specified as a command line option.

Analyses are defined in configuration files that contain any combination and number of available building blocks presented in Section 3. There are three stages to the Analysis Pipeline, which are defined using that configuration file:

In the first stage, each incoming flow record is tested against each of the **filters** that the user has defined. These filters are similar to the `rwfilter` command line. The records that pass each filter are handed to each association interested in the those particular flow records. Filters are described in Section 3.2.

In the second stage, evaluations and statistics process the records: **Evaluations** compare internal state to a user defined threshold. **Statistics** compute state values and then export that state based on a user-defined interval of time. See Section 3.4 for descriptions of evaluations and statistics.

The last stage is the alerting stage. It checks the evaluations and statistics to see if there are any alerts to be sent. This alerting stage also checks with named lists that are configured to periodically send their entire contents as alerts.

To assist in entering data and sharing data among multiple filters, the Analysis Pipeline allows the administrator to create a list. A **list** can reference an existing SiLK IPset file, contain values entered directly into the configuration file, or be created by a mechanism inside `pipeline` itself.

Filters, evaluations, statistics, and lists are all built independently of each other, with each having a unique name. They are linked together by referencing those unique names.

Any number of evaluations and statistics can receive the records held by each filter. However, evaluations and statistics can only have one filter providing flow records to it. Filters delete their flows after each flow file, or group of flows from a socket, as the other stages of Pipeline take what they need from the flows during processing.

An additional concept in the Analysis Pipeline is an **internal filter**. Internal filters can be used to store intermediate “successes” based on flow records and filters. These “successes” are not major enough to yield individual alerts, but can shape future processing. Internal filters are used when the analysis is more complex than simply passing a filter and transferring data to evaluations, and they allow for multistage analysis: “Flow record A met criteria, and future flow records will be combined with record A for more in-depth analysis.”

Another use case for internal filters is when the user wants to build a list containing a particular field (or list of fields) from all of the records that meet the criteria of a certain filter. “Build a list of all IP addresses that send out web traffic from port 80.”

## 1.1 Using Schemas

Starting with version 5, `pipeline` uses schemas from the SchemaTools library to describe the data. Earlier versions of `pipeline` had a priori knowledge of the data that was coming (SiLK IPv4 records). Version 5.0 runs with built-in schemas for SiLK records using both IPv4 and IPv6. If SiLK is configured to handle IPv6, a schema will be built for each. If it is not installed, only an IPv4 schema will be used. If after installation, SiLK is recompiled with IPv6, or upgraded to a version with IPv6, Pipeline will need to be reconfigured, built, and installed, so it will include the IPv6 schema.

As a result, `pipeline` will be able to handle records of either type, including SiLK files that contain both types of records. The two schemas are identical up to the point of dealing with IP addresses. As a result, if an evaluation is calculating “SUM BYTES”, it will be able to run on both record types. If something uses a particular type of IP address (v4 or v6), that filter/evaluation/statistic will only process those types of records.

Versions 5.1 and beyond include a schema builder specifically for YAF data, and one that dynamically generates schemas based on the contents of IPFIX templates. Based on command line options, `pipeline` will know which schema builder to request schemas from.

IPFIX uses information elements (IEs). Each IE has an enterprise ID, and an element ID. Standard, public elements have an enterprise ID equal to 0. The standard IPFIX elements with their names, IDs, and descriptions can be found at <http://www.iana.org/assignments/ipfix/ipfix.xhtml>.

If, regardless of the fields used, you wish to restrict the processing to one particular record type, you can specify a schema to use for the filter/eval/stat by adding the schema name, or schema number after the filter/eval/stat name. See sections 3.2 and 3.4 for more information.

## 1.2 Configuration File Organization

To specify the filters, evaluations, statistics, and lists, a configuration language is used. The configuration information can be contained in a single file, or it may be contained in multiple files that are incorporated into a master file using `INCLUDE` statements. Syntax:

```
INCLUDE "path-name"
```



Multiple levels of file `INCLUDE` statements are supported. Often the top level configuration file is named `pipeline.conf`, but it may have any name.

Examples:

```
INCLUDE "/var/pipeline/filters.conf"
INCLUDE "evaluations.conf"
```

Filters, evaluations, and statistics can appear in any order in the configuration file(s) as long as each item is defined before it is used. The only exception is named lists being referenced by filters. These can be referenced first, and defined afterwards. Since filters are used by evaluations and statistics, it is common to see filters defined first, then finally evaluations and statistics, with list configurations at the end.

In the configuration file, blank lines and lines containing only whitespace are ignored. Leading whitespace on a line is also ignored. At any location in a line, the octothorp character (a.k.a. hash or pound sign, `#`) indicates the beginning of a comment, which continues until the end of the line. These comments are ignored.

Each non-empty line begins with a command name, followed by zero or more arguments. Command names are a sequence of non-whitespace characters (typically in uppercase), not including the characters `#` or `"`. Arguments may either be textual atoms (any sequence of alphanumeric characters and the symbols `_`, `-`, `@`, and `/`), or quoted strings. Quoted strings begin with the double-quote character (`"`), end with a double-quote, and allow for C-style backslash escapes in between. The character `#` inside a quoted string does not begin a comment, and whitespace is allowed inside a quoted string. Command names and arguments are case sensitive.

Every filter, evaluation, statistic, and list must have a name that is unique within the set of filters, evaluations, statistics, or lists. The name can be a double-quoted string containing arbitrary text or a textual atom.

When `pipeline` is invoked, the `--configuration` switch must indicate the file containing all of the configuration information needed for the Analysis Pipeline.

To assist with finding errors in the configuration file, the user may specify the `--verify-configuration` switch to `pipeline`. This switch causes `pipeline` to parse the file, report any errors it finds, and exit without processing any files.

To print the contents of the arriving schemas, and also as a way to verify the viability of the data source configuration, the information elements that are available for processing can be displayed by specifying the `--print-schema-info`.

To print the schema information and verify all configuration files, specify `--show-schema-info-and-verify`.

## 1.3 Configuration Order

The ordering of blocks in the configuration file does have an impact on the data processing of `pipeline`. Comparisons (in filters) and checks (in evaluations) are processed in the order they appear, and to pass the filter or evaluations, all comparisons or checks must return a true value. It is typically more efficient to put the more discerning checks and comparisons first in the list. For example, if you are looking for TCP traffic from IP address 10.20.30.40, it is better to do the address comparison first and the protocol comparison second because the address comparison will rule out more flows than the TCP comparison. This reduces the number of comparisons and in general decreases processing time. However, some comparisons are less expensive than others (for example, port and protocol comparisons are faster than checks against an IPset), and it may reduce overall time to put a faster comparison before a more-specific but slower comparison.

## 1.4 Configuration Syntax and Underscores

All keywords and SiLK field names in `pipeline` are to be entered in capital letters. YAF and IPFIX field names are mixed case.

Throughout `pipeline` documentation and examples, underscores have been used within keywords in some places, and spaces used in others. With the current release, both options are accepted. For example: `ALERT_EACH_ONLY_ONCE` and `ALERT EACH ONLY ONCE` are interchangeable. Even `ALERT_EACH ONLY _ONCE` is allowed. Underscores and spaces will each be used throughout this document as a reminder that each are available for use.

## 1.5 A note on time values

SECONDS, MINUTES, HOURS, and DAYS are all acceptable values for units of time. Combinations of time units can be used as well, such as 1 HOUR 30 MINUTES instead of 90 MINUTES.

## 1.6 A note on this versions beyond version 3

Most configuration files that worked with Analysis Pipeline version 3 will be incompatible with Analysis Pipeline versions 4.x and 5.x. The use of Pipeline version 3.0 was not widespread and we felt the improvements made to the configuration file for versions 4.x and 5.x were worth the potential effort required to convert the files from version 3 to 4/5. The changes are minimal, and we have faith that users can make the conversions on their own. If you are having difficulty getting a version 3 configuration file to work on versions 4 and 5, please contact [netsa-help@cert.org](mailto:netsa-help@cert.org) for assistance.

## Chapter 2

# Installation and Integration

This chapter describes building the Analysis Pipeline application from source code, installing it, and integrating it to work with your data source(s). Before building `pipeline`, you may want to read Section 2.5 below to determine how you will be integrating the Analysis Pipeline with SiLK's packing tools.

**Note:** In the examples presented in this chapter, a leading dollar sign (\$) represents a normal user's shell prompt, indicating a command to be run by the installer that does not require root (administrator) privileges. Commands to be run that (most likely) require root privileges have a leading octothorpe (hash or pound sign, #) as the prompt. A line ending in a backslash (\) indicates the that line has been wrapped for improved readability, and that the next line is a continuation of the current line. A line that begins with neither a dollar sign nor an octothorpe and that is not a continuation line represents the output (i.e., the result) of running a command.

## 2.1 Building pipeline

This section describes how to build and install the `pipeline` daemon.

As of `pipeline` 4.0, the Analysis Pipeline can send alerts using an external library called `libsнарf` which is available on the NetSA web site, <http://tools.netsa.cert.org/>. It is recommended that you download and install `libsнарf` prior to building `pipeline`. Consult the `libsнарf` documentation for details on building, installing, and configuring `libsнарf`. Once you have built and installed `libsнарf`, note the name of the directory containing the `libsнарf.pc` file. This file is normally in `${prefix}/lib/pkgconfig` where `${prefix}` is the `--prefix` value used when `libsнарf` was compiled, typically `/usr` or `/usr/local`.

To build `pipeline`, you will need access to SiLK's library and header files. `pipeline` version 5 and above require at least SiLK-3.0. If you are using an RPM installation of SiLK, be certain the `silk-devel` RPM is installed.

`pipeline` version 5.3 needs `schemaTools` version 1.1 or later. Versions of `schemaTools` can also be downloaded at <http://tools.netsa.cert.org/>. `schemaTools` must be installed before building `pipeline`.

`pipeline` version 5 and above requires `libfixbuf` version 1.4.

Finally, you will need to download a copy of the Analysis Pipeline 5.3.1 source code from the NetSA web site, <http://tools.netsa.cert.org/>. The source code will be in a file named `analysis-pipeline-5.3.1.tar.gz`.

Use the following commands to unpack the source code and go into the source directory:

```
$ tar xzf analysis-pipeline-5.3.1.tar.gz
$ cd analysis-pipeline-5.3.1
```

The `configure` script in the `analysis-pipeline-5.3.1` directory is used to prepare the `pipeline` source code for your particular environment. The rest of this section explains the `configure` command option by option.

The first thing you must decide is the parent directory of the pipeline installation. Specify that directory in the `--prefix` switch. If you do not specify `--prefix`, the `/usr/local` directory is used.

```
configure --prefix=/usr
```

Next, you may use the `--with-libsнарf` switch to specify the directory containing the `libsнарf.pc` file:

```
configure --prefix=/usr --with-libsнарf=/usr/lib/pkgconfig
```

If you do not specify the directory, the `configure` script will attempt to find the `libsнарf.pc` file in directories specified in the `PKG_CONFIG_PATH` environment variable and in the standard locations used by the `pkg-config` tool.

pipeline version 5 needs SchemaTools and Fixbuf. If they were installed using the same `--prefix` that pipeline will use, then nothing needs to be done. If not, there are two switches that need to be added to the configuration:

`--with-fixbuf-prefix` set to the prefix used for fixbuf, or the top level directory where the fixbuf source code is located

`--with-schematools-prefix` set to the prefix used for schemaTools, or the top level directory where the schemaTools source code is located.

If SiLK is configured to handle IPv6, a schema will be built for both IPv4 and IPv6. If IPv6 is not installed, only an IPv4 schema will be used. If after installation, SiLK is recompiled with IPv6, or upgraded to a version with IPv6, Pipeline will need to be reconfigured, built, and installed, so it will include the IPv6 schema.

Finally, you must specify the location of SiLK's header and library files. With SiLK 3.0 and later, the `silk_config` program is available. Integration with SiLK is done by using the `--with-silk-config` switch and specify the path to the `silk_config` program. That program gives information on how SiLK was compiled, and it is installed in `${prefix}/bin/silk_config` where `${prefix}` is the `--prefix` value used when SiLK was compiled, typically `/usr` or `/usr/local`.

For example, assuming you have installed SiLK in `/usr` and you want to install pipeline in `/tools/usr`, specify:

```
configure --prefix=/tools/usr --with-libsнарf=/usr/lib/pkgconfig \
  --with-silk-config=/usr/bin/silk_config
```

### 2.1.1 Compiling and installing

To configure the Analysis Pipeline source code, run the `configure` script with the switches you determined above:

```
$ configure --prefix=/usr --with-libsнарf=/usr/lib/pkgconfig \
  --with-silk-config=/usr/bin/silk_config
```

Once pipeline has been configured, you can build it:

```
$ make
```

To install pipeline, run:

```
# make install
```

Depending on where you are installing the application, you may need to become the root user first.

To ensure that pipeline is properly installed, try to invoke it:

```
$ pipeline --version
```

If your installation of SiLK is not in `/usr`, you may get an error similar to the following when you run `pipeline`:

```
pipeline: error while loading shared libraries: libsilk-thrd.so.2:
cannot open shared object file: No such file or directory
```

If this occurs, you need to set or modify the `LD_LIBRARY_PATH` environment variable (or your operating system's equivalent) to include the directory containing the SiLK libraries. For example, if SiLK is installed in `/usr/local`, you can use the following to run `pipeline`:

```
$ export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
$ pipeline --version
```

To avoid having to specify `LD_LIBRARY_PATH` when running `pipeline`, you can specify the following when you run `make` (this assumes a Linux system):

```
$ env LD_RUN_PATH=/usr/local/lib make
```

## 2.2 Data Source Configuration

Pipeline can handle multiple types of data, and can handle multiple data sources at the same time. If using a single data source, the details can be specified just using command line switches. If using multiple data sources (or just one without using the command line), use the `--data-source` switch to provide a data source configuration file to `pipeline`. Details for the configuration file are in Section 2.2.6.

For each data source to be configured, there are three general aspects of it that need specified: The record type (`silk`, `YAF`, or `ipfix`), the data location (socket, directory polling, or file list), and the timing options. These are discussed in sections below.

Even though YAF exports its information using IPFIX, it has its own switch and schema builder because the deep packet inspection information is nested in lists that are dynamically filled in with templates at runtime. The YAF schema builder knows what elements to expect in what template, and can export those using schemas to `pipeline`. The YAF schema builder also knows what to do with each template that arrives based on the template ID for efficiency reasons.

### 2.2.1 SiLK Records

The `--silk` switch tells `pipeline` to use the SiLK schema builder. Getting SiLK data into `pipeline` is the same as in earlier versions. The first way to get SiLK data into `pipeline` is to have it poll a directory for new flow files. The `--incoming-directory` switch specifies the directory where flows will appear. These files will be deleted after processing, unless the `--archive-directory` is specified, in which case the files will be stored in the given directory. This input method will cause `pipeline` to be run as a daemon, unless `--do-not-daemonize` is specified. The other option is to specify the names of the flow files on the command line. Use the `--name-files` switch to tell `pipeline` to look for the list of files to process on the command line at the end of the switches. These files are NOT deleted after processing. If not using `time-is-clock`, the files should be specified in order based on the timing field used from oldest to newest.

As a sole, or PRIMARY data source, a SiLK data source has to use the flow end time as in versions 4.x, so it cannot have a custom timing source (`--time-is-clock` or a different field), and cannot use `--break-on-recs`.

If a SiLK data source is a secondary data source, and the primary data source uses `--time-is-clock`, the SiLK records will be processed using `--time-is-clock`, just as any other secondary data source would.

To create a SiLK builder using a data source configuration file, use: `SILK BUILDER`.

## 2.2.2 YAF Records

The `--yaf` switch tells `pipeline` to use the YAF schema builder.

YAF data can be received in a list of files on the command line using `--name-files`, or from a tcp or udp socket.

A timing source (2.2.5) must be specified. If a socket is used, `--break-on-recs` must be used as well.

To create YAF builder in the data source configuration file, use: YAF BUILDER.

It's likely that the best way to get records to Pipeline is to add an exporter to the instance of Super Mediator that is connected to the main YAF instance. This will ensure that Pipeline will not interfere with the rest of the collection architecture.

## 2.2.3 Raw IPFIX Records

The `--ipfix` switch tells `pipeline` to use the default ipfix schema builder.

IPFIX data can be recieved using a list of files on the command line, a udp or tcp socket connection, or by polling a directory.

A timing source (2.2.5) must be specified. If a socket is used, `--break-on-recs` must be used as well.

To create IPFIX builder in the data source configuration file, use: IPFIX BUILDER.

## 2.2.4 Data Location

`--name-files` indicates that the names of the files to be processed are listed on the command line at the end of the rest of the switches. To specify `--name-files` using the data source configuration file, use: NAME FILES.

`--incoming-directory` indicates the directory to poll to look for new files to process. To specify `--incoming-directory` using the data source configuration file, use: INCOMING DIRECTORY <DIR\_PATH>

`--archive-directory` indicates the directory to place successfully processed files. Can only be used with `--incoming-directory`, but must be different than the incoming directory. To specify `--archive-directory` using the data source configuration file, use: ARCHIVE DIRECTORY <DIR\_PATH>

`--error-directory` indicates the directory to place files that were not properly processed. Can only be used with `--incoming-directory`. To specify `--error-directory` using the data source configuration file, use: ERROR DIRECTORY <DIR\_PATH>

`--polling-interval` How long a file must be of constant size before it is processed. This ensures that a file is completely written. To specify `--polling-interval` using the data source configuration file, use: POLLING INTERVAL <NUMBER>

`--polling-timeout` How long to wait for new files to be put in the incoming direcorey before it returns "empty" to pipeline. This will cause pipeline to move onto the next data source, or repeat the process if this is the only data source. To specify `--polling-timeout` using the data source configuration file, use: POLLING TIMEOUT <NUMBER>

`--tcp-port` indicates the port to listen on for a tcp connection for a YAF or IPFIX socket connection. To specify `--tcp-port` using the data source configuration file, use: TCP PORT <NUMBER>

`--udp-port` indicates the port to listen on for a udp connection for a YAF or IPFIX socket connection. To specify `--udp-port` using the data source configuration file, use: UDP PORT <NUMBER>

In version 5.1, `pipeline` will only reestablish a socket connection if it is the only active data source. If there are 2 sockets used, the first one that goes down will never be restarted, but the second one will listen again after it goes down. This will be fixed by the next released version.

## 2.2.5 Timing and Flow Grouping Options

Version 4.x of `pipeline` relied on flow end time values from SiLK records to advance `pipeline`'s internal timing. With the expansion of data types, and dynamic schema generation, there is no way to ensure that there is a sufficient timing source in the records. Another aspect taken for granted in earlier versions only using SiLK flow files is knowing when to be done filtering records and run the evaluations and statistic. In the 4.x versions, the end of the flow file provided this break. The options below are used to ensure that `pipeline` has an adequate timing source and knows when to run the evaluations and statistics.

If you are using a SiLK data source, timing has to be done the same as in previous versions, so none of these options are allowed to be combined with a SiLK data source.

`--time-is-clock` tells `Pipeline` to use the internal system clock as the timing source, rather than values from the records like earlier versions. If there are analytic specific time windows used, this is not a good option for testing (with `--named-files`), as processing will happen faster than if running in live streaming mode. For example, say it takes 1 second to process a file with 5 minutes worth of network traffic. If there are 10 files processed with `--time-is-clock`, `pipeline`'s internal time will advance 10 seconds versus 50 minutes, which could throw off time window expectations. To specify `--time-is-clock` using the data source configuration file, use: `TIME IS CLOCK`.

If there is a timing source in the record, such as YAF's `flowEndMilliseconds`, it can be specified either by name, or by the {enterprise id, element number} tuple. This information element must be of type `DATETIME_(MILLI|MICRO|NANO| )SECONDS`, `UNSIGNED_64` if the units field is `(MILLI|MICRO|NANO)SECONDS`, or `UNSIGNED_32` if the units field is `SECONDS`.

`--time-field-name` is used to specify the name of the information element to be used as the timing source. To specify `--time-field-name` using the data source configuration file, use: `TIMING FIELD NAME <fieldNameString>`

`--time-field-ent` and `--time-field-id` can be combined to specify the enterprise id and element number of the timing element. To specify `--time-field-ent` and `--time-field-id` using the data source configuration file, use: `TIMING ENT <NUMBER>` and `TIMING ID <NUMBER>`.

The final option for a timing source is for it to come from the schema. A schema builder can specify that a particular element can be used as a timing source. `--time-from-schema` tells `pipeline` to get the timing element from the schema.

`--break-on-recs` tells `pipeline` how many records to process before breaking to run the evaluations. This feature is not permitted when using a SiLK data source. It is required when the data source uses a socket connection. It is optional when reading YAF or IPFIX files. If used in conjunction with non-SiLK files, `pipeline` will break when the record threshold is hit, and also at the end of the file. To specify `--break-on-recs` using the data source configuration file, use: `BREAK ON RECS <NUMBER>`.

There can also be timing issues when using multiple data sources that have no way to guarantee their times are in sync. In the data source configuration file, required for multiple data sources, one is declared the primary data source, and the rest are secondary. Time will advance during `pipeline` processing when records from the primary data source are received. When records from other data sources are processed, `pipeline`'s internal time is not advanced. If an information element is specified as the timing source, but it is not contained in the schema that describes the record, time does not advance, just as it would not for a secondary data source. However, if `pipeline` is using the internal clock to advance the time, then time will advance no matter the source of the data records.

## 2.2.6 Data Source Configuration File

When using multiple data sources, a configuration file is required to instantiate them.

The `--data-source` switch is used to specify that configuration file. If this switch is used, all other switches described above cannot be used as everything must be specified in this configuration file.

The format of this configuration file is similar to the one used for filters, evaluations, etc. Below is an example using each data source and each input type. Any other keywords that aren't used in this example are listed afterwards. Capitalized words are keywords, lowercase words, such as data source names, are up to the user.

```
PRIMARY DATA SOURCE yafSocket
  YAF BUILDER
  TCP PORT 18000
```

```

    BREAK ON RECS 10000
    TIME IS CLOCK
END DATA SOURCE

SECONDARY DATA SOURCE silkPolling
    SILK BUILDER
    INCOMING DIRECTORY "/data/pipelineIncoming"
    ERROR DIRECTORY "/data/pipelineError"
    ARCHIVE DIRECTORY "/data/pipelineArchive"
    FLAT ARCHIVE
END DATA SOURCE

SECONDARY DATA SOURCE ipfixFiles
    IPFIX BUILDER
    NAME FILES #a list of filenames will be on the command line
    TIMING FIELD NAME flowEndMilliseconds
END SECONDARY DATA SOURCE

```

#### Other options

```

POLLING INTERVAL <integer seconds> #the amount of time that a file has to be
of constant size, meaning it's done being written to.
POLLING TIMEOUT <integer seconds> #how long to wait for new files before
returning empty to Pipeline so it can move to the next data source.
UDP PORT <integer port> Port to listen on for a udp connection
TIMING ENT <integer enterprise id> IPFIX enterprise ID for the timing field
TIMING ID <integer element id> IPFIX element ID for the timing field

```

## 2.3 Stats Log Updates

pipeline logs periodic usage statistics. There are two parts to the usage updates. The first part contains the number of seconds since the last update, the number of records processed since the last update, and the number of files read since the last update. If using a socket connection and `--break-on-recs`, the number of breaks takes the place of the number of files. If there are multiple data sources, the number of records and files processed is logged per data source in addition to the overall counts.

The second part of the usage update includes the number of bytes of memory used by each evaluation and statistic.

Each item in the update has a label, followed by a colon, then a value. Items are pipe delimited.

The interval for the status log updates is set by using `--stats-log-interval` on the command line. The values for this option are in minutes. The default interval is 5 minutes. Setting this value to zero will turn off this feature. The daemon config variable is `STATS_LOG_INTERVAL`.

## 2.4 Preparing to run

Once pipeline is installed, you must perform configuration before running the Analysis Pipeline. (Before performing the steps in this section, you should read Section 2.6 on automating the starting and stopping of the pipeline daemon, since settings in the two sections are related.)

The pipeline application requires access to the `SiLK country_codes.pmap` file. That file is used to map IP addresses to two letter country code designations. You may use the `country_codes.pmap` file from your existing SiLK installation. If you do not have the `country_codes.pmap` file, instructions on building it are provided in the `rwgeoip2ccmap` manual page and in the SiLK



Installation Handbook. The `country_codes.pmap` file should be installed in `${prefix}/share/silk/country_codes.pmap`, where `${prefix}` is the location where SiLK was installed (often `/usr` or `/usr/local`). You may also specify the file's location in the `SILK_COUNTRY_CODES` environment variable, or via `pipeline`'s `--country-code-file` command line switch.

In this section, we use `/var` as the directory where daemons store their run-time files, and `/etc` as the location for configuration files. The directory layout presented here is merely a suggestion. You may configure the files and directories however you wish, and they may vary depending on your operating system and your organization's policies and practices.

Create the following directories:

**`/etc/pipeline`** holds configuration subdirectories.

**`/etc/pipeline/config`** holds configuration files that are specified on the `pipeline` command line.

**`/etc/pipeline/watchlists`** holds binary SiLK IPset files used by watch lists and other filters.

**`/var/pipeline`** holds run-time subdirectories.

**`/var/pipeline/incoming`** is where files for `pipeline` to process are dropped.

**`/var/pipeline/error`** is where `pipeline` stores any files it cannot process.

**`/var/pipeline/log`** holds `pipeline`'s log files.

If you are using `pipeline` for watchlist detection, solicit the IP lists from your analysts. For each separate watchlist, use SiLK's `rwsetbuild` tool to create a binary SiLK IPset containing the IPs on that watchlist. Put each IPset file into the `/etc/pipeline/watchlist` directory. Use the watchlist examples presented in Section 4.3 to create the `pipeline`'s configuration file(s), making sure to use the complete path name to the IPset files. Store the configuration file(s) in `/etc/pipeline/config`.

The remainder of this chapter assumes the top-level configuration file has the name `pipeline.conf`, and that that file is in the `/etc/pipeline/config` directory.

To load the schema builder and print their contents, illuminating their information elements available for use, run

```
$ pipeline --print-schema-info
```

To print the schema information and verify the configuration file syntax, run

```
$ pipeline \
  --configuration=/etc/pipeline/config/pipeline.conf \
  --show-schema-and-verify
```

To just verify the syntax of the configuration file, run

```
$ pipeline \
  --configuration=/etc/pipeline/config/pipeline.conf \
  --verify-config
```

If `pipeline` does not run and reports an error similar to "cannot open shared object file", set the `LD_LIBRARY_PATH` environment variable as described above.

If there are errors in the `pipeline.conf` file, correct the errors and try again.

The following assumes the Analysis Pipeline was built with `libsнарf` support, and it invokes `pipeline` as a daemon:

```
# pipeline \
  --configuration=/etc/pipeline/config/pipeline.conf \
  --log-dir=/var/pipeline/log \
  --incoming-dir=/var/pipeline/incoming \
  --error-dir=/var/pipeline/error
```

This command will cause `pipeline` to look for incremental files in `/var/pipeline/incoming`. Files that are successfully processed will be deleted. (If you want to keep files after `pipeline` finishes with them, provide the `--archive-directory` switch.) If `pipeline` encounters an error processing a file, the file will be moved to the `/var/pipeline/error` directory. `pipeline` will write its log messages to `/var/pipeline/log/pipeline-DATE.log`, and these files will be rotated at midnight.

If this command fails because `pipeline` cannot file the `country_codes.pmap` file (which maps IP addresses to country codes), see the beginning of this section for instructions on installing this file.

### 2.4.1 Alerting with libsnarf

When the Analysis Pipeline is built with support for libsnarf, the `SNARF_ALERT_DESTINATION` environment variable is set to tell `pipeline` the address where a `snarfd` process is listening for alerts. The environment variable takes the form `tcp://HOST:PORT` which specifies that the `snarfd` process is listening on `HOST` at `PORT`.

Instead of specifying the `SNARF_ALERT_DESTINATION` environment variable, you may specify the location using the `--snarf-destination` switch.

When neither `SNARF_ALERT_DESTINATION` nor `--snarf-destination` is specified, `pipeline` prints the alerts encoded using JSON (JavaScript Object Notation). The JSON output is written to `pipeline`'s normal log file.

### 2.4.2 Legacy Alerting

If the Analysis Pipeline was built without libsnarf support, the alerts generated by `pipeline` are written to local files. The location of the alert files must be specified using the `--alert-log-file` and `--aux-alert-file` switches on the `pipeline` command line. Assuming the directory layout described above, one would add the following switches to the command line specified above:

```
--alert-log-file=/var/pipeline/log/alert.log
--aux-alert-file=/var/pipeline/log/auxAlert.log
```

Alerted information is split between the two files in the following way:

1. The record that generated the alert will be converted to text format and appended to the file `/var/pipeline/log/alert.log`.
2. Information about the data metrics used to trigger an alert, the information stored for a statistic, and the `FOREACH` value used to identify a bin of state data will be appended to `/var/pipeline/log/auxAlert.log`. The contents of lists will be sent here as well.

The main reason for this separation of alert files is that if `pipeline` is being used for watchlisting, the main alert file will contain the records that hit the watchlist. This allows for easier processing of alerts that only contain records, versus records combined with data metric information.

You should configure the `logrotate` program to rotate both alert files daily. Unlike the other log files that `pipeline` creates, this file is **not** rotated automatically by the `pipeline` daemon. To configure `logrotate` under Linux, create a new file named `pipeline` in `/etc/logrotate.d`, and use the following as its contents:

```
/var/pipeline/log/alert.log {
    missingok
    compress
    notifempty
    nomail
    rotate 1
    daily
}
```

The `/var/pipeline/log/alert.log` file contains pipe-delimited (`|`-delimited) text. This text can be read by a security information management (SIM) system such as ArcSight. The Analysis Pipeline includes the file `pipeline.sdkfilereader.properties` that can be used as a starting point to create a new ArcSight Log File FlexConnector that will monitor that `alert.log` file.

To use ArcSight, customize the `pipeline.sdkfilereader.properties` file and place a copy of the file (with the same filename) in the agent configuration directory on the machine running the ArcSight connector, `CONNECTOR_HOME/current/user/agent/flexagent`. If necessary, contact your ArcSight representative for instructions on how to get the Connector installation wizard. When prompted for the type of SmartConnector to install, select the entry for “ArcSight FlexConnector File”.

## 2.5 Integrating with SiLK packing

Normally the Analysis Pipeline application, `pipeline`, runs as a daemon during SiLK’s collection and packing process. `pipeline` runs on the flow records after they have been processed by `rwflowpack`, since `pipeline` may need to use the class, type, and sensor data that `rwflowpack` assigns to each flow record.

`pipeline` should get a copy of each incremental file that `rwflowpack` generates. Where to install `pipeline` so that it sees every file will depend on how you configured SiLK’s packing system. If you are using one of the daemons `rwsender`, `rwreceiver`, or `rwflowappend` downstream of `rwflowpack`, then integrating `pipeline` is straightforward. If none of these daemons are in use at your site, you must modify how `rwflowpack` runs.

The remainder of this section describes each approach.

### 2.5.1 Using `rwsender`

To use `pipeline` with the `rwsender` in SiLK-2.2 or later, specify a `--local-directory` argument to `rwsender`, and have `pipeline` use that directory as its `incoming-directory`, for example:

```
# rwsender ... --local-directory=/var/pipeline/incoming ...  
  
# pipeline ... --incoming-directory=/var/pipeline/incoming ...
```

### 2.5.2 Using `rwreceiver`

When the Analysis Pipeline is running on a dedicated machine separate from the machine where `rwflowpack` is running, one can run `rwsender` on the machine where `rwflowpack` is running and have it send the incremental files to a dedicated `rwreceiver` on the machine running `pipeline`. In this case, the `incoming-directory` for `pipeline` will be the `destination-directory` for `rwreceiver`. For example:

```
# rwreceiver ... --destination-dir=/var/pipeline/incoming ...  
  
# pipeline ... --incoming-directory=/var/pipeline/incoming ...
```

When `pipeline` is running on a machine where an `rwreceiver` (version 2.2. or newer) is already running, one can specify an additional `--duplicate-destination` directory to `rwreceiver`, and have `pipeline` use that directory as its `incoming directory`. For example:

```
# rwreceiver ... --duplicate-dest=/var/pipeline/incoming ...  
  
# pipeline ... --incoming-directory=/var/pipeline/incoming ...
```

### 2.5.3 Using rflowappend

One way to use pipeline with rflowappend is to have rflowappend store incremental files into an archive-directory, and have pipeline process those files. When using rflowappend 2.3.0 or newer, specify the `--flat-archive` switch which causes rflowappend to place the files into the root of the archive-directory. For this situation, make the archive-directory of rflowappend the incoming-directory of pipeline:

```
# rflowappend ... --flat-archive --archive-dir=/var/pipeline/incoming

# pipeline ... --incoming-directory=/var/pipeline/incoming ...
```

Older versions of rflowappend store the incremental files in subdirectories under the archive-directory. For this case, you must specify a `--post-command` to rflowappend to move (or copy) the files into another directory where pipeline can process them. For example:

```
# rflowappend ... --archive-dir=/var/rflowappend/archive \
  --post-command='mv %s /var/pipeline/incoming' ...

# pipeline ... --incoming-directory=/var/pipeline/incoming ...
```

### 2.5.4 Using rflowpack only

If none of the above daemons are in use at your site because rflowpack writes files directly into the data repository, you must modify how rflowpack runs so it uses a temporary directory that rflowappend monitors, and you can then insert the call to pipeline after rflowappend has processed the incremental files.

Assuming your current configuration for rflowpack is:

```
# rflowpack --sensor-conf=/var/rflowpack/sensor.conf \
  --log-directory=/var/rflowpack/log \
  --root-directory=/data ...
```

You can modify it as follows:

```
# rflowpack --sensor-conf=/var/rflowpack/sensor.conf \
  --log-directory=/var/rflowpack/log \
  --output-mode=sending \
  --incremental-dir=/var/rflowpack/incremental \
  --sender-dir=/var/rflowappend/incoming ...

# rflowappend --root-directory=/data \
  --log-directory=/var/rflowappend/log \
  --incoming-dir=/var/rflowappend/incoming \
  --error-dir=/var/rflowappend/error \
  --flat-archive \
  --archive-dir=/var/pipeline/incoming

# pipeline --incoming-directory=/var/pipeline/incoming \
  --error-directory=/var/pipeline/error \
  --log-directory=/var/pipeline/log \
  --configuration-file=/etc/pipeline/config/pipeline.conf
```

If you are using a version of SiLK older than 2.3.0, change the `--flat-archive` switch on rflowappend to be `--post-command='mv %s /var/pipeline/incoming'`.

## 2.6 Automating the Analysis Pipeline

When Pipeline is consuming more than just a list of files, it can be run as a daemon. If run from the command line using a socket or an incoming directory to provide data, it will turn itself into a daemon automatically unless `--do-not-daemonize` is specified. If running Pipeline as a daemon with `service pipeline start`, you must use a data source configuration file to specify data sources and input types, even if there is only a single source. This is specific in the daemon config file by with the `DATA_SOURCE_CONFIG_FILE` variable.

To provide easier control of the pipeline daemon in UNIX-like environments, an example control script (an `sh`-script) is provided. This control script will be invoked when machine is booted to start the Analysis Pipeline, and it is also invoked during shutdown to stop the Analysis Pipeline. Use of the control script is optional; it is provided as a convenience.

As part of its invocation, the control script will load a second script that sets shell variables the control script uses. This second script has the name `pipeline.conf`. Do not confuse this variable setting script (which follows `/bin/sh` syntax) with the `/etc/pipeline/config/pipeline.conf` configuration file, which is loaded by the pipeline application and follows the syntax described in Chapter 3.

If you are using an RPM installation of pipeline, installing the RPM will put the control script and the variable setting script into the correct locations under the `/etc` directory, and you can skip to the variable setting section below.

If you are not using an RPM installation, the `make install` step above installed the scripts into the following location relative to pipeline's installation directory. You will need to copy them manually into the correct locations.

**`share/analysis-pipeline/etc/init.d/pipeline`** is the control script. Do not confuse this script with the pipeline application.

**`share/analysis-pipeline/etc/pipeline.conf`** is the variable setting script used by the control script.

Copy the control script to the standard location for start-up scripts on your system (e.g., `/etc/init.d/` on Linux and other SysV-type systems). Make sure it is named `pipeline` and has execute permissions. Typically, this will be done as follows:

```
# cp ${prefix}/share/analysis-pipeline/etc/init.d/pipeline \
    /etc/init.d/pipeline
# chmod +x /etc/init.d/pipeline
```

Copy the variable setting script file into the proper location. Typically, this will be done as follows:

```
# cp ${prefix}/share/analysis-pipeline/etc/pipeline.conf \
    ${prefix}/etc
```

Edit the variable setting script to suit your installation. Remember that the variable setting script must follow `/bin/sh` syntax. While most of the variables are self-explanatory or can be derived from the documentation elsewhere in this chapter and pipeline's manual page, a few variables deserve some extra attention:

**ENABLED** Set this variable to any non-empty value. It is used by the control script to determine whether the administrator has completed the configuration.

**CREATE\_DIRECTORIES** When this value is yes, the control script creates any directories that the daemon requires but are nonexistent.

**SILK\_LIB\_DIR** The directory holding the `libsilk.so` file. This may be needed if `libsilk.so` is not in a standard location understood by the system linker. (Set this if you must set `LD_LIBRARY_PATH` to run pipeline.) See the `ld.so(8)` and `ldconfig(8)` manual pages for more details.

**COUNTRY\_CODES** The location of the country code map, if it cannot be found in the standard location.

**LOG\_TYPE** The daemons support writing their log messages to the `syslog(3)` facility or to local log files rotated at midnight local time. Set this to “syslog” to use syslog, or to “legacy” to use local log files. (The setting here does **not** affect the `alert.log` file, since it is handled differently.)

**LOG\_DIR** When the LOG\_TYPE is legacy, the logging files are written to this directory. The `/var/log` directory is often used for log files.

**PID\_DIR** The daemons write their process identifier (PID) to a file in this directory. By default this variable has the same value as LOG\_DIR, but you may wish to change it. On many systems, the `/var/run` directory holds this information.

**USER** The control script switches to this user (see `su(1)`) when starting the daemon. The default user is `root`. Note that the Analysis Pipeline can be run as an ordinary user.

At this point you should be able to use the control script as follows to start or stop the `pipeline`:

```
# /etc/init.d/pipeline start
# /etc/init.d/pipeline stop
```

To automate starting and stopping the `pipeline` when the operating system boots and shuts down, you need to tell the machine about the new script. On RedHat Linux, this can be done using:

```
# chkconfig --add pipeline
```

(If you have installed `pipeline` from an RPM, you do not need to perform this step.)

At this point, you should be able to start the `pipeline` using the following command:

```
# service pipeline start
```

## Chapter 3

# Configuration Language

### 3.1 Fields and Field Lists

All fields in the data records can be used to filter data, along with some derived fields. With this version, the fields are not set ahead of time like in past versions. The available fields are based on the data sources and the schemas contained therein. If a SiLK data source is used, the fields will be the same as in previous versions, but now with IPv6 addresses available if the SiLK installation has enabled it.

Available fields can be combined into tuples, e.g. {SIP, DIP}, for more advanced analysis. These tuples are represented in the configuration file by listing the fields with spaces between them. When processed, they are sorted internally, so SIP DIP SPORT is the same as SPORT DIP SIP.

SchemaTools uses IPFIX information elements, so each element in a schema has an enterprise id and an element id attached to it. Some are part of the general set, with enterprise id equal to 0, and other are custom elements with a non zero enterprise id. In past versions of pipeline, the name of the element or field was the only way to identify elements. In version 5.1 and beyond, the {enterprise, id} tuple can be used to identify elements. This can be used to avoid confusion if names are different in different schemas. For example, for backwards compatibility, the SiLK builder still uses the name "SIP", while YAF and other IPFIX based records will use "sourceIPv4Address". If {0,8} is used, pipeline won't care what the name of the element is in the schema.

With each specific element listed below, the enterprise id, element id tuple will be included in curly brackets after the name. Enterprise id 6871 is the enterprise id for CERT/SEI, so our custom elements will have this enterprise id.

SiLK data can still use "ANY IP", as described below in the SiLK section. In addition to the legacy groups of elements (IP and PORT), schemaTools provides each schema groups that can be used with ANY based on their IPFIX type value. The values are: OCTET\_ARRAY, UNSIGNED\_8, UNSIGNED\_16, UNSIGNED\_32, UNSIGNED\_64, SIGNED\_8, SIGNED\_16, SIGNED\_32, SIGNED\_64, FLOAT\_32, FLOAT\_64, BOOLEAN, MAC\_ADDRESS, STRING, IPV4\_ADDRESS, IPV6\_ADDRESS. Groups will only be included in a schema, and thus available to pipeline users as long as the group is non empty.

Beginning with version 5.3, "ANY" fields can be used anywhere in the configuration file that regular fields are allowed. This includes, but is not limited to derived fields, FOREACH, PROPORTION, DISTINCT, FILTERS, OUTPUT LISTS, FIELD BOOLEANS, and INTERNAL FILTERS. "PAIR" fields can be used anywhere a 2-tuple can.

To see which elements will be available for the given data, the command line switch `--print-schema-info` can be used. In addition to a list of available elements, a list of groups and their contents will be printed.

For the dynamically generated schemas from the YAF and IPFIX data sources, the list of available information elements is only created after a connection to the data has been made. This can be either a socket connection with initial data transmission, opening a file from the command line, or the first file being read from a directory being polled. If there are multiple data sources, pipeline will wait to connect to the primary, and then go in order based on the configuration file connecting to the other sources. Pipeline will not begin processing data records from any source, without having connected to all sources, as pipeline does not process the main configuration file without knowledge of the available elements.

### 3.1.1 Fields with Multiple Values

Pipeline can handle data in lists, notably the DPI information in YAF records. It can also handle the situation where there are repeated fields in a record. In either case where there can be multiple values for a particular field name, Pipeline will process each value. These types of fields may also be referred to as "loopable fields".

#### Loopable Fields in Filters

When using a loopable field in a comparison, if ANY of the values for the field meet the criteria of the filter, it will return true. For example, if the filter is implementing a DNS watchlist, if any of the `dnsQName` values in the record are in the watchlist, the entire record is marked as passing the filter.

#### Loopable Fields as Foreach

When a loopable field is chosen as the FOREACH field, there will be a state listing for every value in the record for that field.

#### Loopable Fields in a Primitive

If a loopable field is used as the field for a primitive, all values will be used. For example, if computing the SUM of a loopable, all of the values will be included in the sum.

### 3.1.2 SiLK Records

IP addresses and ports have directionality, source and destination. The keyword ANY can be used to indicate that the direction does not matter, and both values are to be tried. The ANY \* fields can go anywhere inside the field list, the only restrictions are that the ANY must immediately precede IP, IPv6, PORT, and that there can only be one ANY in a field list.

The available fields are:

**ANY IP** Either the source address or destination address. This applies to IPv4.

**ANY IPv6** Same ANY IP, but with SIP\_V6 DIP\_V6.

**IP PAIR** Either the {SIP, DIP} tuple or the {DIP, SIP} tuple. This has changed in versions 5.3 and beyond. The ANY has been removed from referencing pairs due to internal parsing issues.

**IPv6 PAIR** Either the {SIP\_V6, DIP\_V6} tuple or the {DIP\_V6, SIP\_V6} tuple. This has changed in versions 5.3 and beyond. The ANY has been removed from referencing pairs due to internal parsing issues.

**ANY\_PORT** Either the source port or destination port.

**PORT PAIR** Either the {SPORT, DPORT} tuple or the {DPORT, SPORT} tuple. This has changed in versions 5.3 and beyond. The ANY has been removed from referencing pairs due to internal parsing issues.

**APPLICATION {6871, 33}** The *service* port of the record as set by the flow generator if the generator supports it, or 0 otherwise. For example, this would be 80 if the flow generator recognizes the packets as being part of an HTTP session.

**BYTES {6871, 85}** The count of the number of bytes.

**BYTES PER PACKET {6871, 106}** An integer division of the bytes field and the packets field. It is a 32-bit number. The value is set to 0 if there are no packets.

**CLASSNAME {6871, 41}** The class name assigned to the record. Classes are defined in the `silk.conf` file.

**DIP {0, 12}** The destination IPv4 address.



**DIP\_V6 {0, 28}** The destination IPv6 address.

**DPORT {0, 11}** The destination port.

**DURATION {0, 161}** The duration of the flow record, in integer seconds. This is the difference between the ETIME and STIME.

**END\_SECONDS {0, 151}** The wall clock time when the flow generator closed the flow record in seconds.

**ETIME {0, 153}** The wall clock time when the flow generator closed the flow record in milliseconds.

**FLAGS {6871, 15}** The union of the TCP flags on every packet that comprises the flow record. The value can contain any of the letters F, S, R, P, A, U, E, and C. (To match records with either ACK or SYN|ACK set, use the IN\_LIST operator.) The flags formatting used by SiLK can also be used to specify a set of flags values. S/SA means to only care about SYN and ACK, and of those, only the SYN is set. The original way Pipeline accepted flags values, the raw specification of flags permutation is still allowed.

**ICMPCODE {0, 177}** The ICMP code. This test also adds a comparison that the protocol is 1.

**ICMP\_TYPE {0, 176}** The ICMP type. This test also adds a comparison that the protocol is 1.

**INITFLAGS {6871, 14}** The TCP flags on the first packet of the flow record. See FLAGS.

**INPUT {6871, 10}** The SNMP interface where the flow record entered the router. This is often 0 as SiLK does not normally store this value.

**NHIP {0, 15}** The next-hop IP of the flow record as set by the router. This is often 0.0.0.0 as SiLK does not normally store this value.

**NHIP\_V6 {0, 62}** See NHIP above, but the IPv6 version.

**OUTPUT {6871, 11}** The SNMP interface where the flow record exited the router. This is often 0 as SiLK does not normally store this value.

**PACKETS {6871, 86}** The count of the number of packets.

**PROTOCOL {0, 4}** The IP protocol. This is an integer, where 6 is TCP.

**SENSOR {6871, 31}** The sensor name assigned to the record. Sensors are defined in the `silk.conf` file.

**SESSIONFLAGS {6871, 16}** The union of the TCP flags on the second through final packets that comprise the flow record. See FLAGS.

**SIP {0, 8}** The source IPv4 address.

**SIP\_V6 {0, 27}** The source IPv6 address

**SPORT {0, 7}** The source port.

**START\_SECONDS {0, 150}** The wall clock time when the flow generator opened the flow record in seconds.

**STIME {0, 152}** The wall clock time when the flow generator opened the flow record in milliseconds.

**TCP\_STATE {6871, 32}** any combination of the letters F, T, or C, where

- F** indicates the flow generator saw additional packets in this flow following a packet with a FIN flag (excluding ACK packets)
- T** indicates the flow generator prematurely created a record for a long-running connection due to a timeout.
- C** indicates the flow generator created this flow as a continuation of long-running connection, where the previous flow for this connection met a timeout

**TYPENAME {6871, 30}** The type name assigned to the record. Types are defined in the `silk.conf` file.

Derived fields available for use with STIME and ETIME include: HOUR, DAY\_OF\_WEEK, DAY\_OF\_MONTH, and MONTH.

FLOW\_KEY\_HASH is also available as all of the fields are guaranteed to be in SiLK flow records.

### 3.1.3 YAF Records

YAF is capable of creating flow records used by SiLK, so most of the SiLK elements are available in YAF, though they use the standard IPFIX element names.

Element names and numbers of the potential fields in the core YAF record are listed in the YAF documentation at <http://tools.netsa.cert.org/yaf/yaf.html>, in the section labeled "OUTPUT: Basic Flow Record".

In addition to core flow fields, YAF exports information from deep packet inspection in dynamic lists. These are added to the schema as virtual elements from the yaf schema builder. Those elements are listed by name, element numbers, and are clearly described in the yaf online documentation at: <http://tools.netsa.cert.org/yaf/yafdpi.html>

As of version 5.3, there are three DPI fields exported by YAF that have had their information element changed by Pipeline to disambiguate them from fields in the core part of the records.

Two of these changes are with DNS records. When YAF exports the IP address returned in a DNS response, it uses `sourceIPv4Address` and `sourceIPv6Address`. To keep those separate from the source IP addresses used in the core of the records, those IP fields are changed to used fields named `rrIPv4` and `rrIPv6` for IPv4 and IPv6 respectively.

The other field changed from YAF export to Pipeline processing is the `protocolIdentifier` field in the DNSSEC DNSKEY record. This field has been changed to a field named `DNSKEY_protocolIdentifier` to keep it separate from the `protocolIdentifier` in the core flow record exported by YAF.

Certain fields are in the standard YAF record and can almost certainly be used. Derived fields available for use with `flowStartMilliseconds` and `flowEndMilliseconds` include: HOUR, DAY\_OF\_WEEK, DAY\_OF\_MONTH, and MONTH.

FLOW\_KEY\_HASH is also available as all of the fields are part of the standard YAF record.

If YAF is exporting DPI elements, depending on their contents, the remainder of the derived fields can be used.

This table contains elements in the core YAF records that are always present. The names are case sensitive. Even if a field in this list has a corresponding SiLK field, these names and numbers must be used. This list is taken from the YAF documentation in the OUTPUT section at: <http://tools.netsa.cert.org/yaf/yaf.html> Associated SiLK fields are in parentheses where applicable. Reverse fields are not listed in this table as they are not guaranteed to be present.

**flowStartMilliseconds {0,152}** Flow start time in milliseconds (STIME)

**flowEndMilliseconds {0,153}** Flow end time in milliseconds, good choice for timing source to mimic SiLK processing. (ETIME)

**octetTotalCount {0,85}** Byte count for the flow.(BYTES)

**packetTotalCount {0,86}** Packet count for the flow.(PACKETS)

**sourceIPv4Address {0,8}** Source IP address of the flow.(SIP)

**destinationIPv4Address {0,12}** Destination IP address of the flow.(DIP)

**ANY IPV4\_ADDRESS** Group of elements that are IP addresses, which will contain `sourceIPv4Address` and `destinationIPv4Address`.(ANY IP)

**IPV4\_ADDRESS PAIR** Group of elements that are IP addresses, which will contain `sourceIPv4Address` and `destinationIPv4Address`.(IP PAIR)

**sourceIPv6Address {0,27}** Source IPv6 address of the flow. (SIP\_V6)

**destinationIPv6Address {0,28}** Destination IPv6 address of the flow.(DIP\_V6)

**ANY IPV6\_ADDRESS** Group of elements that are IPv6 addresses, which will contain `sourceIPv6Address` and `destinationIPv6Address`.(ANY IPv6)

**IPV6\_ADDRESS PAIR** Group of elements that are IPv6 addresses, which will contain `sourceIPv6Address` and `destinationIPv6Address`.(ANY IPv6)

**sourceTransportPort {0,7}** Source TCP or UDP port of the flow. (SPORT)

**destinationTransportPort {0,11}** Destination TCP or UDP port of the flow.(DPORT)

**flowAttributes {6871,40}** Bit 1, all packets have fixed size. Bit 2, out of sequence

**protocolIdentifier {0,4}** IP protocol of the flow.(PROTOCOL)

**flowEndReason {0,136}** 1: idle timeout. 2: active timeout 3: end of flow 4: force end 5: lack of resources

**silkAppLabel {6871,33}** Application label, if YAF is run with `-applabel`.(APPLICATION)

**vlanId {0,58}** VLAN tag of the first packet

**ipClassOfService {0,5}** For IPv4, the TOS field, for IPv6, the traffic class

**tcpSequenceNumber {0,184}** Initial TCP sequence number

**initialTCPFlags {6871,14}** TCP flags of initial packet. (INITFLAGS)

**unionTCPFlags {6871,15}** Union of the TCP flags of all packets other than the initial packet.(SESSIONFLAGS)

### 3.1.4 IPFIX Records

IPFIX records can include any element available. There are no elements that can be assumed. You have to know the data to know or use the switch `--print-schema-info`

The various derived fields are available based on the fields on the IPFIX records.

### 3.1.5 PMAPs

Prefix Maps (pmaps) are part of the SiLK tool suite and can be made using `rwpmmapbuild`. Their output can be used just like any other field in pipeline. It can make up part of a tuple, be used in FOREACH, and be used in filtering. PMAPs take either an IP address, or a PROTOCOL PORT pair as inputs.

Before use, pmaps must be declared in the configuration file. The declaration line is not part of a FILTER or EVALUATION, so it is by itself, similar to the INCLUDE statements. The declaration line starts with the keyword PMAP, followed by a string for the name without spaces, and lastly, the filename in quotation marks.

```
PMAP userDefinedFieldName "pmapFilename"
```

Now that the PMAP is declared, the field name can be used throughout the file. Each time the field is used, the input to the pmap must be provided. This allows different inputs to be used throughout the file, without redeclaring the pmap.

```
userDefinedFieldName(inputFieldList)
```

For each type of pmap, there is a fixed list of inputFieldLists:

#### IP Address pmaps

SIP Use the SIP as the key to the pmap.

DIP Use the DIP as the key to the pmap.

ANY IP Use the SIP, then use the DIP as keys to the pmap.

#### Protocol Port pair pmaps

```

PROTOCOL SPORT Use the PROTOCOL SPORT tuple as the key to the pmap
PROTOCOL DPORT Use the PROTOCOL DPORT tuple as the key to the pmap
PROTOCOL ANY PORT Use the PROTOCOL SPORT tuple, then the PROTOCOL DPORT tuple as keys to the pmap.

```

The examples above use the names of the fields from SiLK for simplicity. Any field of type IPV4\_ADDRESS or IPV6\_ADDRESS can be used for an IP pmap. The IPFIX elements for SIP and DIP are sourceIPv4Address and destinationIPv4Address.

The port-protocol pmaps must use the IPFIX elements of: protocolIdentifier (in place of PROTOCOL above), sourceTransportPort, and destinationTransportPort (in place of SPORT and DPORT above).

Below is an example that declares a pmap, then filters based on the result of the pmap on the SIP, then counts records per pmap result on the DIP.

```

PMAP thePmapField "myPmapFile.pmap"

FILTER onPmap
    thePmapField(SIP) == theString
END FILTER

STATISTIC countRecords
    FILTER onPmap
    FOREACH thePmapField(DIP)
    RECORD COUNT
END STATISTIC

```

### 3.1.6 Field Booleans

Field booleans are custom fields that consist of an existing field and a list of values. If the value for the field is in the value list, then the field boolean's value is TRUE. These are defined similar to PMAPs, but use the keyword FIELD BOOLEAN. For example, to define a boolean named webPorts, to mean the source port is one of [80, 8080]:

```
FIELD BOOLEAN sourceTransportPort webPorts IN [80, 8080]
```

Now, webPorts is a field that can be used anywhere in the configuration file that checks whether the sourceTransportPort is in [80, 8080].

If used in filtering, this is the same as just saying: sourceTransportPort IN LIST [80, 8080].

However, if used as a part of FOREACH, the value TRUE or FALSE will be in the field list, to indicate whether the sourceTransportPort is 80 or 8080.

Another example could be a boolean to check whether the hour of the day, derived from a timestamp, is part of the work day. There could be a statistic constructed to report byte counts binned by whether the hour is in the workday, which is 8am to 5pm in this example.

```

FIELD BOOLEAN HOUR_OF_DAY(flowStartSeconds) workday IN
    [8,9,10,11,12,13,14,15,16,17]

STATISTIC workdayByteCounts
    FOREACH workday
    SUM octetTotalCount
END STATISTIC

```

### 3.1.7 DNS Derived Fields

If the records include fields containing domain names, the following fields can be used. If used on non DNS string, there will not be an error when parsing the configuration file, but most will not return data as DNS dot separators are required for processing.

The field to be operated on is put in parentheses after the derived field name.

These fields can be used anywhere in a pipeline configuration file like any other field.

**DNS\_PUBLIC\_SUFFIX(dnsFieldName)** A STRING field that contains the public suffix of the domain name. The default public suffix list is Mozilla's. A user can specify a different list with the `--dns-public-suffix-file`.

**DNS\_SLD(dnsFieldName)** A STRING field that contains the second level domain from a domain name.

**DNS\_HOST(dnsFieldName)** A STRING field that contains the host name from a domain name, simply, the string before the first dot separator.

**DNS\_PRIVATE\_NAME(dnsFieldName)** A STRING field that contains every part of the domain name but the public suffix.

**DNS\_SLD+TLD(dnsFieldName)** A STRING field that contains the second level domain and the top level domain.

**DNS\_INVERT(dnsFieldName)** A STRING field that reverses the order of the labels of the presumed dns string. It handles escaped dots. It also ignores starting or ending dots.

**DNS\_NORMALIZE(dnsFieldName)** A STRING field that is the original dnsFieldName converted to lower case, and removal of any starting and / or ending dots. "WeIrD.dns.COM." normalizes to: "weird.dns.com"

Derivation of fields can be nested as well, such as:

```
DNS_SLD(DNS_INVERT(DNS_NORMALIZE(dnsQName)))
```

For the following example domain name: tools.netsa.cert.org

```
DNS_PUBLIC_SUFFIX is "org"
DNS_SLD is "cert"
DNS_HOST is "tools"
DNS_PRIVATE_NAME is "tools.netsa.cert"
DNS_SLD+TLD is "cert.org"
DNS_INVERT is org.cert.netsa.tools
```

### 3.1.8 Timestamp Derived Fields

These derived fields pull out human readable values from timestamps. The values they pull are just integers, but in filters, pipeline can accept the words associated with those values, e.g. JANUARY is translated to 0, as is SUNDAY. These fields work with field types: DATETIME\_SECONDS, DATETIME\_MILLISECONDS, DATETIME\_MICROSECONDS, DATETIME\_NANOSECONDS. Each will be converted to the appropriate units for processing. The system's timezone is used to calculate the HOUR value.

The field to be operated on is put in parentheses after the derived field name.

These fields can be used anywhere in a pipeline configuration file like any other field.

**HOUR(timestampFieldName)** The integer value for the hour of the day where midnight is 0 and 11pm being 23.

**DAY\_OF\_WEEK(timestampFieldName)** The integer value of the day of the week where SUNDAY is 0. The text names of the days in all capital letters are accepted by the configuration file parser as values for filtering.

**DAY\_OF\_MONTH(timestampFieldName)** The integer value of the day of the month, where the first day of the month is 1.

**MONTH(timestampFieldName)** The integer value of the month of the year where JANUARY is 0. The text names of the months in all capital letters are accepted by the configuration file parser as values for filtering.

### 3.1.9 Other Derived Fields

The field to be operated on is put in parentheses after the derived field name.

These fields can be used anywhere in a pipeline configuration file like any other field.

**STRLEN(stringFieldName)** An integer that is the length of the string field.

**quotedString SUBSTR\_OF(stringFieldName)** A boolean value as to whether the quoted string is a substring of the stringFieldName value. For example: "www" SUBSTR\_OF(dnsQName) will return TRUE if "www" is in the domain name.

**FLOWKEYHASH** A 32-bit integer that is the flow key hash from the flow record. No fields need specified as it is a standard calculation. Using this as a filter can be helpful in batch mode when trying to isolate a particular flow.

All derived fields can use ANY fields, such as:

```
STRLEN(ANY STRING)
```

## 3.2 Filters

The Analysis Pipeline passes each flow record through each filter to determine whether the record should be passed on to an evaluation or statistic.

A filter block starts with the **FILTER** keyword followed by the name of the filter, and it ends with the **END FILTER** statement. The filter name must be unique across all filters. The filter name is referenced by evaluations, internal filters, and statistics.

Filters are initially marked internally as inactive, and become active when an evaluation or statistic references them.

Filters are composed of **comparisons**. In the filter block, each comparison appears on a line by itself. If **all** comparisons in a filter return a match or success, the flow record is sent to the evaluation(s) and/or statistic(s) that use the records from that filter.

If there are no comparisons in a filter, the filter reports success for every record.

Each comparison is made up of three elements: a field, an operator, and a compare value, for example **BYTES > 40**. A comparison is considered a match for a record if the expression created by replacing the field name with the field's value is true.

### 3.2.1 Operators and Compare Values

Eight operators are supported. The operator determines the form that the compare value takes. Hexadecimal values can be used as compare values in filters wherever integers can be used.

**IN\_LIST** Used to test whether a record's field is included in the given list. The compare value can be a list that was previously defined by an evaluation or internal filter, an IPSet filename, or defined in-line:

1. The contents of the list can be entered directly into the configuration file. The elements are comma-separated, surrounded by square brackets, [ and ]. As an example, the following matches FTP, HTTP, and SSH traffic in the filter:

```
DPORT IN_LIST [21, 22, 80]
```

2. Bracketed lists can also be used to enter tuples of information directly into a filter. This is done using nested bracketed lists. One caveat is that this is the one case that the ordering of the fields in the field list matters (which is required to keep the list values in the order that pipeline arranges the fields during processing). The fields must follow this ordering schema: SIP, DIP, SPORT, DPORT, PROTOCOL, STIME, DURATION, TYPENAME, CLASSNAME, SENSOR, ENDTIME, INITFLAGS, RESTFLAGS, TCPFLAGS, TCP\_STATE, APPLICATION, INPUT, OUTPUT, PACKETS, BYTES, NHIP, ICMPTYPE, ICMPCODE, ANY IP, ANY PORT, BYTES PER PACKET. An example is filtering for sip 1.1.1.1 with sport 80, and 2.2.2.2 with sport 443:

```

FILTER sipSportPair
    SIP SPORT IN LIST [[1.1.1.1, 80],[2.2.2.2,443]]
END FILTER

```

3. `pipeline` can take a number of formats for files with lists of values. The filename must be in double quotes. Syntax:

```
fieldList IN LIST "/path/to/watchlist.file"
```

If the `fieldList` consists of one field and if it is of type `IPV4_ADDRESS` or `IPV6_ADDRESS`, the file **MUST** be a `SiLK IPSet`. A `fieldList` of just an IP cannot be any of the types described below.

A file can be used to house both types of bracketed lists described above, both the single and double bracketed lists. This has to be formatted exactly as if it was typed directly into the config file. The format is such that a user should be able to copy and paste the contents of files in this format into the config file and vice versa. The single line (there cannot be any newline characters in the list) of the bracketed list must have a new line at the end.

If the `fieldList` consists of a single field, a simple watchlist file can be used to hold the values. This format requires one value per line. The format of each value type is the same as if it was typed into the configuration file. Comments can be used in the file by setting the first character of the line to "#". The value in the field being compared against the watchlist must be an exact match to an entry in the file for the comparison to be true.

The exact match requirement can cause problems for DNS fields. `pipeline` has no way to know that a particular field value is a DNS domain name string, such that it would return a match for "www.url.com" if "url.com" was in the list. To overcome this deficiency, a watchlist can put a particular string on the first line to tell `pipeline` to process this watchlist as a DNS watchlist. The first line of the file must be: "###format:dns". It must be the first line of the file. When processing the file, `pipeline` will normalize the field value, making it all lower case, and removing starting or ending dots.

4. If there is a single field in the `fieldList`, and if that is an IP address, this bracketed list can contain `IPSet` files mixed with IP addresses that will all be combined for the filter:

```
SIP IN LIST ["/data/firstIPset.set", 192.168.0.0/16, "/data/secondIPset.set"]
```

5. The name of a list that is filled by the outputs of an evaluation, or an internal filter. The tuple in the filter must entirely match the tuple used to fill the list.

```
SIP DIP PROTO SPORT DPORT IN LIST createdListOfFiveTuples
```

6. The only way to use a logical OR with filters is to create a full filter for describing the sets of conditions you'd like to OR together. For such a filter, the field is `FLOW RECORD`. For example, to do TCP sport 80 OR UDP dport 23.

```

FILTER tcp80
    SPORT == 80
    PROTOCOL == 6
END FILTER
FILTER udp23
    DPORT == 23
    PROTOCOL == 17
END FILTER
FILTER filterUsingTcp80OrUdp23
    FLOW RECORD IN LIST [tcp80,udp23]
END FILTER

```

**NOT\_IN\_LIST** Same as `IN_LIST`, but succeeds if the value is not in the list.

**==** Succeeds when the value from the record is equal to the compare value. This also encompasses IPv4 subnets. For example, the following will succeed if either the source or destination IP address is in the 192.168.x.x subnet:

```
ANY_IP == 192.168.0.0/16
```

**!=** Succeeds when the value from the record is not equal to the compare value.

**<** Succeeds when the value from the record is strictly less than the compare value.

**<=** Succeeds when the value from the record is less than or equal to the compare value.

- > Succeeds when the value from the record is strictly greater than the compare value.
- >= Succeeds when the value from the record is greater than or equal to than the compare value.

The compare value can reference another field on the flow record. For example, to check whether the source and destination port are the same, use:

```
SPORT == DPORT
```

### 3.2.2 Schema Specification for Filters

As described above, Pipeline version 5 uses schemas to describe how the incoming data is structured. By default, filters are run on all data records that contain the fields necessary for processing. To restrict a filter to only handle records from particular schemas, list schema names or numbers after declaring the name of the filter. If the schema name has spaces, you must put the name in quotes.

```
IPv4 SiLK records are in the schema named: "SiLK IPv4 Schema", number: 5114
IPv6 SiLK records are in the schema named: "SiLK IPv6 Schema", number: 5116
```

To limit a filter to only v4 records:

```
FILTER myV4Filter "SiLK IPv4 Schema"
...
END FILTER
```

or

```
FILTER myV4Filter 5114
...
END FILTER
```

To limit a filter to only v6 records:

```
FILTER myV6Filter "SiLK IPv6 Schema"
...
END FILTER
```

or

```
FILTER myV6Filter 5116
...
END FILTER
```

### 3.2.3 Filter Examples

Looking for traffic where the destination port is 21:

```
FILTER FTP_Filter
  DPORT == 21
END FILTER
```

Watchlist checking whether the source IP is in a list defined by the IPset “badSourceList.set”:



```
FILTER WatchList-BadSourcesList
  SIP IN_LIST "badSourceList.set"
END FILTER
```

Compound example looking for an IP on a watch list communicating on TCP port 21:

```
FILTER watchListPlusFTP
  SIP IN_LIST "badSourceList.set"
  DPORT == 21
  PROTOCOL == 6
END FILTER
```

Look for records with a dns query name with second level domain of "cert"

```
FILTER certSLDs
  DNS_SLD(dnsQName) == "cert"
END FILTER
```

### 3.3 Internal Filters and Named Lists

There are two places where named lists can be created and populated so they can be used by filters: Internal Filters and Output Lists, which are discussed in Section 3.6.2.

In each case, a field list is used to store the tuple that describes the contents of the data in the list. A filter can use these lists if the tuple used in the filters perfectly matches the tuple used to make the list.

#### 3.3.1 Internal Filter Description

An internal filter compares the incoming flow record against an existing filter, and if it passes, it takes some subset of fields from that record and places them into a named list. This list can be used in other filters. There can be any number of these lists.

Internal filters are different from output lists, because they put data into the list(s) immediately, so the contents of the list(s) can be used for records in the same flow file as the one that causes data to be put into the list(s). Output lists, populated by evaluations, are only filled, and thus take effect, for the subsequent flow files, because additions to the list(s) are made after the filters are finished running.

Internal filters are immediate reactions to encountering a notable flow record.

The fields to be pulled from the record and put into the list can be combined into any tuple. The "ANY" fields and "PAIRS" cannot be used here. Details on how to create an internal filter for specific use for the HIGH\_PORT\_CHECK primitive is discussed below.

#### 3.3.2 Internal Filter Syntax

An internal filter is a combination of filters and lists, so both pieces need to be specified in the syntax. A key aspect of the internal filter declaration is to tell it which fields pulled from records that pass the filter, get put into which list. There can be more than one field-list combination per internal filter.

A timeout value must be added to each statement which declares the length of time a value can be considered valid.

```
Syntax:
  INTERNAL_FILTER name of this internal filter
    FILTER name of filter to use
```

```

        fieldList list name timeout
    ...
END INTERNAL FILTER

```

Examples (given an existing filter to find records to or from watchlist)

```

INTERNAL_FILTER watchlistInfo
    FILTER watchlistRecords
    SPORT DPORT watchlistPorts 1 HOUR
    SIP DIP SPORT DPORT PROTOCOL watchlistFiveTuples 1 DAY
END INTERNAL_FILTER

```

This internal filter pulls {SPORT,DPORT} tuples and puts them into a list called watchlistPorts, and those values stay in the list for 1 hour. It also pulls the entire five tuple from those records and puts then into a list called watchlistFiveTuples that stay in the list for 1 DAY.

HIGH\_PORT\_CHECK requires the use of internal filters as they scan for flow records to compare against that can be in the same flow file. The field list for each of these lists are keywords, that in addition to indicating the fields to be stored, tells pipeline how to store them. The keyword is WEB\_REDIR\_LIST.

## 3.4 Introduction to Evaluations and Statistics

Evaluations and statistics comprise the second stage of the Analysis Pipeline. Each evaluation and statistic specifies the name of a filter which feeds records to the evaluation or statistic. Specific values are pulled from those flow records, aggregate state is accumulated, and when certain criteria are met alerts are produced.

To calculate and aggregate state from the filtered flow records, pipeline uses a concept called a **primitive**. These will be described in Section 3.5.

Evaluations are based on a list of checks that have primitives embedded in them. The aggregate state of the primitive is compared to the user defined threshold value and alerts are generated.

Statistics use exactly one primitive to aggregate state. The statistic periodically exports all of the state as specified by a user-defined interval.

If a statistic is utilizing FOREACH, and the state for a particular unique value bin is empty, the value will not be included in an alert for the statistic. A statistic without FOREACH, will output the state value even if it is zero.

An evaluation block begins with the keyword EVALUATION followed by the evaluation name. Its completion is indicated by END EVALUATION.

Similarly, a statistic block begins with the keyword STATISTIC and the statistic's name; the END STATISTIC statement closes the block

The remainder of this section describes the settings that evaluations and statistics have in common, and the keywords they share. A description of primitives is presented in the section 3.5, which hopefully will make the details of evaluations (Section 3.6) and statistics (Section 3.7) easier to follow.

Each of the following commands go on their own line.

### 3.4.1 Id

Each evaluation and statistic must have a unique string identifier. It is placed immediately following the EVALUATION or STATISTIC declaration:

```

EVALUATION myUniqueEvaluationName
...

```

```
END EVALUATION

STATISTIC myUniqueStatisticName
...
END STATISTIC
```

### 3.4.2 Schema Specification for Evaluations

As described above, Pipeline version 5 uses schemas to describe how the incoming data is structured. By default, evaluations and statistics are run on all data records that contain the fields necessary for processing. To restrict an evaluation or statistic to only handle records from particular schemas, list schema names or numbers after declaring the name of the evaluation or statistic. If the schema name has spaces, you must put the name in quotes. Only one schema can be specified here.

```
IPv4 SiLK records are in the schema named: "SiLK IPv4 Schema", number: 5114
IPv6 SiLK records are in the schema named: "SiLK IPv6 Schema", number: 5116
```

To limit an evaluation to only v4 records:

```
EVALUATION myV4Eval "SiLK IPv4 Schema"
...
END EVALUATION
```

or

```
EVALUATION myV4Eval 5114
...
END EVALUATION
```

To limit an evaluation to only v6 records:

```
EVALUATION myV6Eval "SiLK IPv6 Schema"
...
END EVALUATION
```

or

```
EVALUATION myV6Eval 5116
...
END EVALUATION
```

The same format is used for statistics as well.

### 3.4.3 Alert Type

The alert type is an arbitrary, user-defined string. It can be used as a general category to help when grouping or sorting the alerts. If no alert type is specified, the default alert type for evaluations and statistics is “Evaluation” and “Statistic”, respectively.

The value for the alert type does not affect pipeline processing.

Syntax:

```
ALERT TYPE alert-type-string
```

### 3.4.4 Severity

Evaluations and statistics can be assigned a severity level which is included in the alerts they generate. The levels are represented by integers from 1 to 255. The severity has no meaning to the Analysis Pipeline; the value is simply recorded in the alert. The default severity is 1.

The value for the severity does not affect pipeline processing.

Syntax:

```
SEVERITY integer
```

### 3.4.5 Filter Id

Evaluations and statistics (but not file evaluations) need to be attached to a filter, which provides them flow records to analyze. Each can have one and only one filter. The filter's name links the evaluation or statistic with the filter. As a result, the filter must be created prior to creating the evaluation or statistic.

Syntax:

```
FILTER filter-name
```

### 3.4.6 “Binning” by distinct field: FOREACH

Evaluations and statistics can compute aggregate values across all flow records, or they can aggregate values separately for each distinct value of particular field(s) on the flow records—grouping or “binning” the flow records by the field(s). An example of this latter approach is computing something per distinct source address.

FOREACH is used to isolate a value (a malicious IP address), or a notable tuple (a suspicious port pair). The unique field value that caused an evaluation to alert will be included in any alerts. Using FOREACH in a statistic will cause the value for every unique field value to be sent out in the periodic update. There are examples and use cases of this in Section 4.5

The field that is used as the key for the bins is referred to as the unique field, and is declared in the configuration file for the FOREACH keyword, followed by the field name:

```
FOREACH field
```

Any of the fields can be combined into a tuple, with spaces between the individual field names. The more fields included in this list, the more memory the underlying primitives need to keep all of the required state.

To keep state per source IP Address:

```
FOREACH SIP
```

To keep state per port pair:

```
FOREACH SPORT DPORT
```

As with filtering, the ordering of the fields in the tuple does not matter as they are sorted internally.

Starting with version 5.3, ANY fields can be used with FOREACH:

```
FOREACH ANY IP
```

Using FOREACH ANY will cause state to be created for every field in the ANY group specified, regardless of direction. In the FOREACH ANY IP example, state will be created for the SIP value, and also for the DIP value.

There are some limits on which fields can be used, as some evaluations require a particular field(s) be used, and some primitives do not support binning by a field.

File evaluations do not handle records, so the FOREACH statement is illegal.

### 3.4.7 Active status

Normally, evaluations and statistics are marked as active when they are defined. Specifying the INACTIVE statement in the evaluation or statistic block causes the evaluation or statistic to be created, but it is marked inactive, and it will not be used in processing records. For consistency, there is also an ACTIVE statement.

Syntax:

```
INACTIVE
```

### 3.4.8 General Evaluation and Statistic Layouts

```
EVALUATION evalName
  FILTER filterName
  [FOREACH fieldList]
  CHECK statement
    primitive statement
    [TIME WINDOW statement]
  END CHECK
  evaluation specific statements
END EVALUATION
```

```
STATISTIC statName
  FILTER filterName
  [FOREACH fieldList]
  primitive statement
  [TIME WINDOW statement]
  statistic specific statements
END STATISTIC
```

## 3.5 Primitives

Primitives are what pipeline uses to calculate and aggregate state from the filtered flow records. They are the building blocks for evaluations and statistics. Statistics use only one primitive and periodically export the state based on a user-defined interval. Evaluations pair primitives and thresholds and send alerts when the aggregate state of a primitive meets the threshold requirement. Evaluations embed a primitive in a check, and there can be multiple checks whose values are "anded" together to produce an overall answer as to whether the evaluation succeeded, and an alert should be sent.

Each primitive is based on a field from a flow record from which it extracts a value to be aggregated. What the primitive does with this value is based on the type of primitive (outlined below).

The primitive's state can be aggregated based on all of the records, or can be divided into bins based on the value of the user defined field in the flow records. A typical example of this is keeping track of something per source IP address. This feature helps to identify IP addresses or ports involved in anomalous activity. (Mainly for ports and IPs, but works with all flow record fields). The field that

is used to create the bins is referred to as the unique field, and is declared in an evaluation or statistic in the configuration file using the FOREACH command, followed by the field name.

There is a time aspect that affects how data is aggregated. Each primitive is assigned a time window that indicates how long data from each flow record is to be counted in the aggregate state before it is timed out and subtracted. This allows the query of "alert if the count gets to 100 in any 5 minute interval" to be successfully answered. The time window value is given in seconds, minutes, or hours. An infinite time window can be used by specifying the keyword FOREVER.

For each primitive, the syntax for embedding it in a check for an evaluation and in a statistic is listed. When used in evaluations, the arithmetic primitives: RECORD COUNT, SUM, AVERAGE, DISTINCT, and PROPORTION are grouped as threshold checks. Each check starts with the keyword CHECK followed by the type of check. It ends with the keywords END CHECK. Statistics only have one primitive, so they are simpler, and primitives do not need to be embedded in a check.

All of these primitives can be used to build evaluations, but only those specifically labeled can be used to build a statistic. Some primitives have specific requirements, such as being required to be the only one in an evaluation or statistic. These are laid out in each section, along with the memory consumption ramifications for each type. The number of bytes of state that each primitive keeps is listed. If the evaluation or statistic is binning up the state using FOREACH, that number of bytes will be multiplied by the number of unique values seen to get the total memory consumption. If no FOREACH is used, there is only one state value, no multiplier.

Each primitive has certain requirements for information provided, or restrictions on what is allowed. For example, the SUM of SIPs is nonsensical and is not permitted. These will be outlined below.

There may be some aspects of the configuration file that are set automatically by choosing a certain primitive. These will be mentioned below with each primitive when they arise.

Starting with version 5.3, an rwBag can be used to provide custom thresholds for arithmetic primitives. These bags must be used in conjunction with a FOREACH value. This FOREACH field (list) will be the input to the bag to determine the threshold. As a result, the FOREACH field (list) must be 4 bytes or less. It can simply be an IP address, the most likely case, but it could also be SPORT DPORT, as that is a combination of two 2-byte fields, into a 4 byte value. If there is no entry in the bag for a particular FOREACH value, thus the threshold value is non-existent, the flow will be ignored, and no state will be kept.

The syntax is that the integer threshold will be replaced by a quoted string containing the filename of the bag. For example:

```
SUM BYTES > 100
```

Will be replaced by:

```
SUM BYTES > "myBagOfThresholds.bag"
```

### 3.5.1 Time Window

For many primitives, the state is aggregated over a user-specified time window. This window indicates how long data from each flow record is to be counted in the aggregate state before that record's data is timed out and subtracted. This allows the query "Alert if the count gets to 100 in a 5 minute interval" to be successfully answered. The time window is specified with the TIME\_WINDOW keyword followed by a list of number-time-unit pairs. The number may be an integer or a floating-point value. pipeline supports time units of MILLISECONDS, SECONDS, MINUTES, HOURS, or DAYS. For most primitives, any fractional seconds value is ignored. An infinite time window can be specified by using the keyword FOREVER.

This keyword is not required for primitives. Omitting this keyword will cause Pipeline to delete its state after each file is processed. Specifying a time window of 0 is the same as omitting it.

Examples:

```
TIME_WINDOW 6 MINUTES
TIME_WINDOW 4 MINUTES 120 SECONDS # also 6 minutes
TIME_WINDOW 0.1 HOUR              # also 6 minutes
TIME_WINDOW 30 SECONDS
TIME_WINDOW FOREVER
```

Pipeline can base its evaluations on a sliding window, allowing things such as "alert if a SIP sends out more than 10000 bytes in any 5 minute period". That 5 minute period is a sliding time window.

The 5 minutes are measured against "network time". The time is advanced based on the end times in the flows received. If there is a delay in the collection network, causing flows to arrive to pipeline "late", this time window does not get skewed, as it relies on the flows to advance this.

In addition to adding the new flows to the state, evaluations remove expired state (older than the time window), ensuring unwanted, or old, data does not improperly affect the comparison to the threshold.

In an evaluation, the TIME\_WINDOW keyword appears in a CHECK block and applies to that particular primitive. In a statistic, the TIME\_WINDOW command is in main body of the block.

### 3.5.2 Record Count

This primitive type counts the number of records that make it through the filter. It does not pull values from the records, so there is no need to specify a field in the configuration file.

This primitive uses 8 bytes for each state value kept.

#### Record count in a check

```
RECORD COUNT operator threshold
```

This example will send an alert if there are more than 100 records.

```
EVALUATION rcEval
...
    CHECK THRESHOLD
        RECORD COUNT > 100
    END CHECK
...
END EVALUATION
```

#### Record count in a statistic

Statistics do not have thresholds, and this primitive needs no field. This example will generate periodic alerts containing the number of records seen.

```
STATISTIC rcStat
    RECORD COUNT
END STATISTIC
```

### 3.5.3 Sum

This primitive pulls the value of the field specified in the configuration file from a record that passes the filter. These values are added together, and their sum is kept for evaluation. All check parameters are required for this check type.

The available fields for SUM are: BYTES, PACKETS, or DURATION.

This primitive uses 8 bytes for each state value kept.

### Sum in a check

*SUM field operator threshold*

This example will generate an alert if the sum of BYTES is greater than or equal to 1000.

```
EVALUATION sumEval
  CHECK THRESHOLD
    SUM BYTES >= 1000
  END CHECK
END EVALUATION
```

### Sum in a statistic

Statistics do not have thresholds, so this primitive just needs a field. This example will generate periodic alerts containing the sum of the number of packets seen.

```
STATISTIC sumStat
  SUM PACKETS
END STATISTIC
```

## 3.5.4 Average

The AVERAGE primitive is a combination of the sum and record count primitives: it computes the sum of the named volume field and counts the number of records, such that it can compute an average volume per record.

The available field for AVERAGE are BYTES, PACKETS, DURATION, or BYTES PER PACKET.

It uses 12 bytes for each state value kept.

### Average in a check

*AVERAGE field operator threshold*

This example will generate an alert if the average of BYTES PER PACKET is less than 10.

```
EVALUATION avgEval
  CHECK THRESHOLD
    AVERAGE BYTES PER PACKET < 10
  END CHECK
END EVALUATION
```

### Average in a statistic

Statistics do not have thresholds, so this primitive just needs a field. This example will generate periodic alerts containing the running average of the number of packets seen per flow.

```
STATISTIC avgStat
  AVERAGE PACKETS
END STATISTIC
```



### 3.5.5 Distinct

This primitive tallies the number of unique values of the specified field list that have passed the filter. All check parameters are required for this check type. An example of distinct is: "alert if there are 10 unique DIPs seen, regardless of how many times each DIP was contacted". This primitive can be used for statistics. Any number of fields can be combined to be counted, in a field list (except the ANY fields).

The DISTINCT primitive is memory intensive as it keeps track of each distinct value seen and the time when that value was last seen (so that data can be properly aged). When paired with a FOREACH command, the primitive is even more expensive.

#### Distinct in a check

```
DISTINCT field operator threshold
```

This example will generate an alert if more than 50 DPORTs are seen

```
EVALUATION distinctEval
  CHECK THRESHOLD
    DISTINCT DPORT > 50
  END CHECK
END EVALUATION
```

#### Distinct in a statistic

Statistics do not have thresholds, so this primitive just needs a field. This example will generate periodic alerts containing the number of different {SIP, DIP} tuples seen.

```
STATISTIC distinctStat
  DISTINCT SIP DIP
END STATISTIC
```

### 3.5.6 Proportion

This primitive takes a field and a value for that field. It calculates the percentage of the flows that have that value for the specified field. If numeric values are used for the specified field, those values can be represented by integer or hexadecimal numbers.

The option of when to clear the state is automatically set to NEVER for PROPORTION.

This primitive used 16 bytes per state value kept.

#### Proportion in a check

```
PROPORTION field fieldValue operator threshold PERCENT
```

This example will generate an alert if less than 33 percent of traffic is UDP.

```
EVALUATION propEval
  CHECK THRESHOLD
    PROPORTION PROTOCOL 17 < 33 PERCENT
  END CHECK
END EVALUATION
```

### Proportion in a statistic

Statistics do not have thresholds, so this primitive just needs a field. This example will generate periodic alerts containing the percentage of flows where the source port equals 80.

```
STATISTIC propStat
    PROPORTION SPORT 80
END STATISTIC
```

### 3.5.7 Everything Passes

This primitive does not keep any state, it tells pipeline to simply output all flow records that pass the filter. This primitive is typically used for evaluations that alert on watchlists because the watchlist check itself is done at the filter stage.

It must be the only check used in an evaluation and cannot use FOREACH.

Because there is no state kept, running an evaluation with an EVERYTHING\_PASSES primitive has an insignificant effect on the memory usage.

#### Everything passes in a check

There is no state to keep, so there is no additional information needed.

```
EVALUATION epEval
    CHECK EVERYTHING_PASSES
    END CHECK
END EVALUATION
```

#### Everything passes in a statistic

This primitive cannot be used in a statistic. To have pipeline periodically send out the number of flows that a filter identifies, use the RECORD COUNT primitive in a statistic.

### 3.5.8 Beacon

This primitive looks for beacons using SIP, DIP, DPORT, and PROTOCOL as the unique field. If flows show up with end times spaced out in intervals, longer than the user specified time, the four tuple and the record are put into an alert.

The user must provide the threshold of the minimum number of periodic flows to be seen before an alert is generated. The user must also provide the minimum amount of time for the interval between flows. Lastly, the tolerance for a percent variation in flow intervals for when the beacon intervals are not completely steady must be provided.

Do not enter anything for the FOREACH field, it will be done for you. It is automatically set to never clear state upon success.

Beacon finding is very costly simply due to the number of permutations of the SIP DIP DPORT PROTOCOL tuples, and state is needed for each one.

#### Beacon in a check

```
CHECK BEACON
    COUNT minCount CHECK TOLERANCE integerPercent PERCENT
    TIME WINDOW minimumIntervalTimeVal
END CHECK
```

This example will look for beacons that are defined by the following characteristics: There are at least 5 flows with the same {SIP, DIP, DPORT, PROTOCOL} that arrives at a constant interval plus or minus 5 percent. And that interval must be at least 5 minutes.

```
EVALUATION beaconEval
  CHECK BEACON
    COUNT 5 CHECK_THRESHOLD 5 PERCENT
    TIME WINDOW 5 MINUTES
  END CHECK
END EVALUATION
```

### Beacon in a statistic

The Beacon primitive cannot be used in a statistic.

### 3.5.9 Ratio

This primitive calculates the ratio of outgoing to incoming traffic between a pair of IP addresses, and compares it to the specified threshold.

The direction of the traffic can be determined one of two ways:

1. Beacon List: Given an output list created by a beacon evaluation, if the SIP, DIP tuple of the flow record is in the list, it is an outgoing record, if the reverse tuple is in the list, it's an incoming record. Otherwise, it is ignored. This list is provided within the check block by using

*LIST name of the list created in the beacon evaluation*

2. Class/Type field of flow record: If no list is provided, the class/type value from the flow record is used to determine the direction.

There are three options for the FOREACH value for ratio:

1. SIP DIP: This is the original set up, where it tracks IP pairs where the ratio of outgoing to incoming is greater than the threshold.
2. ANY IP: This tracks the outgoing to incoming ratio for each IP, regardless of direction. This is a way to detect whether a particular IP is involved in more outgoing to incoming traffic, regardless of the IP address on the other end of the flow.
3. not specified: If no FOREACH field is specified, this primitive will track overall outgoing to incoming bytes for all flows seen by pipeline, or those that pass the filter connected to this evaluation.

The threshold must at least be that outgoing rate is greater than the incoming rate. Either OUTGOING or INCOMING keywords can be used, as seen below. If OUTGOING is used, the format is  $X > Y$ . If INCOMING is used, the format is  $Y < X$ .

### Ratio in a check

With the requirement that  $\text{integer1} > \text{integer2}$

```
CHECK RATIO
  OUTGOING integer1 TO integer2
  LIST name of list from beacon          # this line is optional
END CHECK
```

The inside of the check reversed with equivalent results:

```
CHECK RATIO
  INCOMING integer2 TO integer1
  LIST name of list from beacon # optional
END CHECK
```

This example will generate an alert if the outgoing to incoming ratio is greater than 10 to 1, without using a beacon list.

```
EVALUATION ratioEval
  CHECK RATIO
    OUTGOING 10 TO 1
  END CHECK
END EVALUATION
```

### Ratio in a statistic

The ratio primitive cannot be used in a statistic.

## 3.5.10 Iterative Comparison

This primitive has been removed in version 5.3.

## 3.5.11 High Port Check

The HIGH\_PORT\_CHECK detects passive data transfer on ephemeral ports. As an example, in passive FTP, the client contacts the server on TCP port 21, and this is the control channel. The server begins listening on an ephemeral (high) port that will be used for data transfer, and the client uses an ephemeral port to contact the server's ephemeral port. Sometimes there are multiple ephemeral connections. Finally, all the connections are closed. Since flows represent many packets, typically the flow representing the traffic on port 21 is not generated until the entire FTP session is ended. As a result, the flow record for port 21 arrives **after** the flow records for the passive transfers.

To detect passive FTP, pipeline uses an internal list of all high port to high port five-tuples. When pipeline sees the port 21 flow record, it determines whether the IPs on that record appear in a five-tuple in the high port list. If a match is found, the traffic between the high ports is considered part of the FTP session.

Syntax:

```
CHECK HIGH_PORT_CHECK
  LIST list-name
END CHECK
```

High Port Check can only be used in an evaluation as seen below.

When using a HIGH\_PORT\_CHECK check in an EVALUATION, there are several additional steps you must take.

1. The FOREACH value must be set to the standard five tuple. The HIGH\_PORT\_CHECK check will set this value for you, and it will issue an error if you attempt to set it to any other value.
2. The filter that feeds the evaluation should look for traffic on the control channel. For FTP this is TCP traffic on port 21.

```
FILTER ftp-control
  ANY_PORT == 21
  PROTOCOL == 6
END FILTER
```

3. A second filter to match traffic between ephemeral ports is created. For example,

```
FILTER passive-ftp
  SPORT > 1024
  DPORT > 1024
  PROTOCOL == 6
END FILTER
```

4. You must create an `INTERNAL_FILTER` block (see Section 3.3). This block uses the filter created in the previous step, and it must specify a list over pairs of source and destination IP addresses. For example,

```
INTERNAL_FILTER passive-ftp
  FILTER passive-ftp
  SIP DIP high-port-ips 90 SECONDS
END INTERNAL_FILTER
```

The list does not need to be created explicitly; the internal filter will create the list if it does not exist.

5. In the `CHECK` block, specify the name of the list that is part of the `INTERNAL_FILTER`. For example,

```
CHECK HIGH_PORT_CHECK
  LIST high-port-ips
END CHECK
```

Putting that together in the `EVALUATION` block, you have:

```
EVALUATION passive-ftp
  FILTER ftp-control
  INTERNAL_FILTER passive-ftp
  CHECK HIGH_PORT_CHECK
    LIST high-port-ips
  END CHECK
END EVALUATION
```

The `HIGH_PORT_CHECK` check is set to always clear the state upon success. This check uses a large amount of memory as the internal list maintains state for each flow record between two ephemeral ports.

### 3.5.12 High Port Check in a Statistic

The `HIGH PORT CHECK` primitive cannot be used in a statistic.

### 3.5.13 Web Redirection

This functionality has been removed in Pipeline V5.

### 3.5.14 Sensor Outage

An evaluation may operate on an input file as a whole, as opposed to operating on every record. This type of evaluation is called a **file evaluation**. It begins with `FILE_EVALUATION` and the name of the file evaluation being created. It ends with `END FILE_EVALUATION`.

The `FILE_OUTAGE` check only works within a `FILE_EVALUATION`. It alerts if pipeline has not received an incoming flow file from the listed sensor(s) in a given period of time.

The FILE\_OUTAGE check is only allowed if there is a SiLK data source.

Syntax:

```
CHECK FILE_OUTAGE
  SENSOR_LIST sensor-list
  TIME_WINDOW number time-unit
END CHECK
```

The TIME\_WINDOW specifies the maximum amount of time to wait for a new sensor file to appear before alerting. The number can be an integer or a floating-point value. Valid time units are MILLISECONDS, SECONDS, MINUTES, HOURS, or DAYS. Fractional seconds are ignored. There is no default time window, and it must be specified.

The SENSOR\_LIST names the sensors that you expect will generate a new flow file more often than the specified time window. This statement must appear in a SENSOR\_LIST check. There are three forms for the statement:

**SENSOR\_LIST *name-or-id*** Watch for missing files from the sensor whose name or numeric identifier is *name-or-id*.

**SENSOR\_LIST [*name-or-id*, *name-or-id*, ...]** Watch for missing files from the sensors whose names and IDs appear in the list.

**SENSOR\_LIST ALL\_SENSORS** Watch for missing files for all sensors.

Example: Alert if any or the sensors **S0**, **S1**, or **S2** do not produce a flow files within two hours:

```
FILE EVALUATION
  CHECK FILE_OUTAGE
    SENSOR_LIST [S0, S1, S2]
    TIME_WINDOW 2 HOURS
  END CHECK
END FILE EVALUATION
```

Example: Alert if any sensor does not produce flow files within four hours:

```
CHECK FILE_OUTAGE
  SENSOR_LIST ALL_SENSORS
  TIME_WINDOW 4 HOURS
END CHECK
```

### 3.5.15 Difference Distribution

This primitive tracks the difference between subsequent values for a specified field. It uses bins, the number of which is based on the length of the field, to keep track of the distribution of those differences. An 8-bit field has 17 bins, a 16 bit field has 33 bins, 32bit->65 bins, and 64bit->129 bins. The bins themselves are 16-bit numbers.

This primitive can only be used in Statistics. It can be used with any field, and can be combined with FOREACH. Based on the way calculations are done, this primitive can only be used with a single schema. If the data sources are only using 1 schema, nothing needs done, if multiple schemas are used, the schema name or number must be stated after the evaluation name. If the SiLK data source is used, the schemas are 5114 for ipv4, and 5116 for ipv6.

The bin chosen to increment is relative to the middle of the array of bins. If there is no difference in the value, the middle bin is incremented. The bin number relative to the middle uses the following calculation:  $\text{bin number} = (\log[\text{base}2] \text{ of the difference}) + 1$ . If the new value is smaller than the old, then a "negative" bin offset is used, as decreases in the value need to be tracked.

The difference distribution primitive can only be run on data coming from a single schema in version 5. If the desired field is in both the IPv4 and IPv6 flow record schemas, please specify which type of records you'd like to use by putting the schema name or schema

number after the name of the statistic as described in section 1.6.2. The filter that the statistic uses can also be limited by schema, as described in section 1.3.2.

Bin Number	Difference Range
lower bins	Bigger negative differences
-4	-15 - -8
-3	-7 - -4
-2	-3 - -2
-1	-1
0	0
1	1
2	2-3
3	4-7
4	8-15
higher bins	Bigger positive differences

Syntax:

```
DIFF DIST field name
```

### 3.5.16 Fast Flux

This primitive sends an alert if it detects a fast flux network. It builds a connected graph of ASN, domain name, ip address tuples. If the connected graph has at least a certain amount of ANSs, domain names, and ip addresses, based on user defined thresholds, that graph is considered to be a fast flux network. When pipeline alerts, the alert can contain all of the elements of the fast flux network (verbose alerting, which is default), or it can send just the number of ASNs, domain name, and ip addresses (existence alerting).

Fast Flux works with IPV6 values as well. Both the IP address field listed, and the pmap referenced must be IPV6.

This evaluation can only run if there is a pmap providing IP to ASN conversions, along with domain name and resolved IP address fields in the data. The check block in the evaluation must specify the information elements used for each field using the ASN(pmap file name), DNS (query name field), and IP\_FIELD(resource record IP field) keywords. Each also needs the threshold for each to determine whether a connected graph is a fast flux network. The general recommended (though not default) thresholds are 5, 5, 5.

Given the available data sources, there are 2 types of records that pipeline can receive, that contain the right information. The easiest is just a flat ipfix record containing domain names and ip addresses (and whatever else) in the main part of the record. If this is the case specifying the elements and their thresholds is sufficient.

If the YAF data source is used, an additional line is needed in the config file. Because pipeline needs to ensure that the IP and DNS values are pulled from the same resource record embedded in a YAF record, the keyword YAF\_DNS\_RECORD\_TYPE must be used in addition to the other element specifiers. The two options for this are: DNS\_A\_RECORD for IPv4, and DNS\_AAAA\_RECORD for IPv6. Due to some required renaming of elements, when using fast flux with YAF records, the IP field used is either named: rrIPv4 or rrIPv6 instead of YAF's true exporting of: sourceIPv4Address and sourceIPv6Address. This removes the confusion of referencing the sip and dip in the main part of the flow records.

Even though this primitive can't be used with a foreach value, which usually prevents output lists, the IPs and / or DNS values in any discovered fast flux network can be put into named lists. This is done by adding the keyword TO after the element specification, followed by the name of the list. This list can be treated just like any other named list, despite the internal differences described below.

As time passes, these graphs continue to grow and use more memory. The individual values do not timeout, so to prevent runaway memory usage, a node maximum must be provided. This is the count of unique values stored in the graph, not the number of ASN, dns, ip tuples. It is specified by: NODE\_MAXIMUM <integer>. When the maximum is reached, all graphs are completely reset.

If the fast fluxing IP and / or DNS values are put into a list, they are not entirely removed when the NODE\_MAXIMUM is reached. Each Fast Flux list has two lists underneath, one containing any active fast flux values, and one containing the values that were active the last time the node maximum was hit. This ensures that a node continually in an active fast flux network will always be in the list in spite of clearing state when NODE\_MAXIMUM is hit.

Here is an example of fast flux with flat IPFIX records, with verbose alerting, and putting the DNS values into a list:

```
PMAP asn "pmaps/asnPmapForFastFlux.pmap"
...
EVALUATION ipfixFastFluxExample
...
    CHECK FAST FLUX
        IP_FIELD sourceIPv4Address 5
        ASN asn 5
        DNS dnsQName 5 TO myFastFluxDNSs
        NODE MAXIMUM 250000
        VERBOSE ALERTS                #not required, defaults to verbose
    END CHECK
...
END EVALUATION
```

Here is an example of fast flux with YAF records, with existence alerting, and putting the IP values into a list:

```
PMAP asn "pmaps/asnPmapForFastFlux.pmap"
...
EVALUATION yafFastFluxExample
...
    CHECK FAST FLUX
        YAF_DNS_RECORD_TYPE DNS_A_RECORD
        IP_FIELD rrIPv4 5 TO fastFluxYafIPs
        ASN asn 5
        DNS dnsQName 5
        NODE MAXIMUM 250000
        EXISTENCE ALERTS
    END CHECK
...
END EVALUATION
```

Fast flux alerts can be very different depending on if Snarf is used versus the legacy alert files, and whether the evaluation is using existence alerts or verbose alerts. When the fast flux evaluation gets to the alerting stage, there could be multiple networks to alert on.

When using existence alerts, Pipeline will only report the number of IPs, ASNs, and DNS names that make up the fast flux network, not the contents. In the legacy auxiliary alert log file, there will be a separate alert for each fast flux network discovered. The metric value in the alert will be a three tuple of the ip, asn, and dns count. When using snarf, all discovered fast flux networks will be listed in a single alert. There will be three metric fields labelled `pipeline.metric.value.ipcount`, `pipeline.metric.value.asncount`, and `pipeline.metric.value.dnscount`. If there are multiple networks alerted on in this single alert, the counts will be listed as parallel arrays with the first element of each array being tied together as one network, the second entries being connected and so on.

When using verbose alerts, the contents of the fast flux networks are included in the alert. The difference in alerting mechanisms are the same in the existence case, with snarf using three parallel arrays for values, and the legacy auxiliary alert file having a separate entry for each network. The number of values in a particular list is given in parentheses before the curly bracketed list of values for parsing purposes.

## 3.6 Evaluation Specific Detail

This section provides evaluation-specific details, building on the evaluation introduction and aggregate function description provided in the previous two sections (3.4 and 3.5).



Each evaluation block must contain one or more check blocks. The evaluation sends each flow record it receives to each check block where the records are aggregated and tests are run. If every check block test returns a true value, the evaluation produces an output entry which may become part of an alert.

### 3.6.1 Checks

In an evaluation, the check block begins with the CHECK statement which takes as a parameter the type of check. The block ends with the END CHECK statement. If the check requires any additional settings, those settings are put between the CHECK and END CHECK statements.

The FILE\_OUTAGE check must be part of a FILE\_EVALUATION block. All other checks must be part of a EVALUATION block.

### 3.6.2 Outputs

When an evaluation threshold is met, the evaluation creates an **output entry**. The output entry may become part of an alert, or it may be used to affect other settings in the pipeline.

#### Output Timeouts

All information contained in alerts is pulled from lists of output entries from evaluations. These output entries can be configured to time out both to conserve memory and to ensure that information contained in alerts is fresh enough to provide value. The different ways to configure the alerting stage are discussed in section 3.6.3.

One way to configure alerting is to limit the number of times alerts can be sent in a time window. This is a place where the output timeout can have a major effect. If alerts are only sent once a day, but outputs time out after one hour, then only the outputs generated in hour before alerting will be eligible to be included in alerts.

When FOREACH is not used, output entries are little more than flow records with attached threshold information. When FOREACH is used, they contain the unique field value that caused the evaluation to return true. Each time this unique field value triggers the evaluation, the timestamp for that value is reset and the timeout clock begins again.

Take for example an evaluation for network profiling that identifies servers. If the output timeout is set to 1 day, then the list of output entries will contain all IP addresses that have acted like a server in the last day. As long as a given IP address is acting like a server, it will remain in the output list and is available to be included in an alert, or put in a named output list as described in section 3.6.2.

Syntax:

```
OUTPUT TIMEOUT timeval
```

Example:

```
OUTPUT TIMEOUT 1 DAY
```

#### Alert on Removal

If FOREACH is used, pipeline can be configured to send an alert when an output has timed out from the output entries list.

Syntax:

```
ALERT ON REMOVAL
```

This is used in addition to the alert settings below, in the "How much to alert" section.

## Shared Output Lists

When FOREACH is used with an evaluation, the field(s) specified for foreach are the fields stored with the output entry, any of those field values can be put into a named output list. If the unique field is a tuple made up of multiple fields, any subset of those fields can be put into a list. There can be any number of these lists. Lists do not have their own timeout values. Since lists are subsets of the fields for all of the output entries of the evaluation, the OUTPUT TIMEOUT for the evaluation applies to all lists as well. When an output entry times out, the value, or subset of that tuple is removed from all output lists that contain it.

These lists can be referenced by filters, or configured separately, as described in section 3.8.

To create a list, a field list of what the output list will contain must be provided. A unique name for this list must be provided as well.

Syntax:

```
OUTPUT LIST fieldList listName
```

If using FOREACH SIP DIP, each of the following lists can be created

```
OUTPUT LIST SIP listOfSips
OUTPUT LIST DIP listOfDips
OUTPUT LIST SIP DIP listOfIPPairs
```

## Clearing state

Once the evaluation's state has hit the threshold and an output entry has been generated, you may desire to reset the current state of the evaluation. For example, if the evaluation alerts when a count of something gets to 1000, you might want to reset the count to start at 0 again.

```
CLEAR ALWAYS
```

For consistency, you may specify the following:

```
CLEAR NEVER
```

It likely that CLEAR ALWAYS is the appropriate choice, but there is no default selection, so one must be specified.

## Too Many Outputs

There are sanity checks that can be put in place to turn off evaluations that are finding more outputs than expected. This could happen from a poorly designed evaluation or analysis. For example, an evaluation looking for web servers may be expected to find less than 100, so a sanity threshold of 1000 would indicate lots of unexpected results, and the evaluation should be shut down as to not take up too much memory or flood alerts.

Evaluations that hit the threshold can be shutdown permanently, or go to sleep for a specified period of time, and turned back on. If an evaluation is shut down temporarily, all state is cleared and memory is freed, and it will restart as if pipeline had just begun processing.

Syntax:

```
SHUTDOWN MORE THAN integer OUTPUTS [FOR timeval]
```

This example will permanently shutdown if there are more than 1000 outputs.

```
SHUTDOWN MORE THAN 1000 OUTPUTS
```

This example will shutdown for 1 day if there are more than 1000 outputs, then it will resume processing.

```
SHUTDOWN MORE THAN 1000 OUTPUTS FOR 1 DAY
```

### 3.6.3 Alerting settings

Alerting is the final stage of the Analysis Pipeline. When the evaluation stage is finished, and output entries are created, alerts can be sent. The contents of all alerts come from these output entries. These alerts provide information for a user to take action and/or monitor events. The alerting stage in `pipeline` can be configured with how often to send alerts and how much to include in the alerts.

Based on the configurations outlined below, the alerting stage first determines if it is permitted to send alerts, then it decides which output entries can be packaged up into alerts.

There are two main sections to alerts: The flow record that generated the alert, and the data metrics depending on the evaluation or statistic. For SiLK data sources, the entire flow record will be included in the alert because there is only 1 hierarchical level to the record. IPFIX data sources can have lists of elements or sub templates in them. Only the top level will be included in the alert, the list contents will not. YAF records that have DPI information will also only have their top level included in the alert. There is no human readable and machine parsable way to include all of the different levels in a single line of an alert. This applies regardless of whether snarf is installed.

#### Extra Alert Field

If there is a field of interest that is not in the main/top level of the schema, it can still be included in the record portion of the alert. This is done with `EXTRA ALERT FIELD`. For example, to include any `dnsQName` values to an alert when using a YAF data source (with `dnsQName` buried in the DPI data), use:

```
EXTRA ALERT FIELD dnsQName
```

The fields in the DPI portion of a YAF record are marked as "loopable", so there can be multiple values for this field. All of these values will be included at the end of the alert.

#### How often to send alerts

Just because there are output entries produced by an evaluation does not mean that alerts will be sent. An evaluation can be configured to only send a batch of alerts once an hour, or 2 batches per day. The first thing the alerting stage does is check when the last batch of alerts was sent, and determine if sending a new batch meets the restrictions placed by the user in the configuration file.

If it determines that alerts can be sent, it builds an alert for each output entry, unless further restricted by the next section that affects how much to alert.

```
ALERT integer-count TIMES number time-units
```

where the number is an integer value, and the time-units can be `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS`, or `DAYS`. Fractional seconds are ignored.

This configuration option does not affect the number of alerts in an alert batch during a time period. It affects the number of times the batches of alerts can be sent per time period. That is why the configuration command says "alert N times per time period", rather than "send N alerts per time period", while the semantic differences are subtle, it has a great affect on what gets sent out.

To have pipeline send only 1 batch of alerts per hour, use:

```
ALERT 1 TIMES 1 HOUR
```

Using `ALERT 2 TIMES 1 HOUR` does NOT mean that the first alert will be sent at time 0, and the second at time 30 minutes. It will send an alert at time 0, then alert the next time there is anything to send out, then will be locked until an hour has passed since the first alert.

To indicate that pipeline should alert every time there are output entries for alerts, use:

`ALERT ALWAYS`

### How much to alert

The second alert setting determines how much information to send in each alert. You may wish to receive different amounts of data depending on the type of evaluation and how often it reports. Consider these examples:

- An evaluation is generating a list of web servers and reporting that list once an hour. You want to get the complete list every hour (that is, in every alert).
- A beacon detection evaluation reports each beacon as soon as it finds the beacon. For this evaluation, you only want to get the beacons found since the previous alert.
- A particular evaluation produces a great deal of output. For this evaluation, you only want to receive the alerts generated in the most recently processed file.
- An evaluation repeatedly finds the same outputs (maybe servers?), but what is notable is when a new one is found. You may only want to hear about each server one time, unless it stops acting like a server, then reestablishes itself.

The amount of data to send in an alert is relevant only when the `OUTPUT_TIMEOUT` statement (Section 3.6.2) includes a non-zero timeout and multiple alerts are generated within that time window.

To specify how much to send in an alert, specify the `ALERT` keyword followed by one of the following:

**EVERYTHING** Package all outputs in the output field list into the current alert.

**SINCE\_LAST\_TIME** Package all of the outputs found since the last alert was sent into the current alert.

**EACH ONLY ONCE** Include each unique value (set with `FOREACH`) in an alert one time only. If the entry in the output table times out, then it rediscovered, the "ONLY ONCE" aspect will be reset, and a new alert will be sent.

The default is `SINCE_LAST_TIME`. If using an `EVERYTHING PASSES` evaluation, be sure to use `ALERT EVERYTHING` to ensure flows from files that arrive with less than a second between them are included in alerts.

The last option is to have an evaluation do its work, but to never send out alerts. If the goal of an evaluation is just to fill up a list so other pieces of pipeline can use the results, individual alerts may not be necessary. Another case is that the desired output of filling these lists is that the lists send alerts periodically, and getting individual alerts for each entry is not ideal. In these cases, instead of the options described above use:

`DO NOT ALERT`

## 3.7 Statistic Specific Detail

Section 3.4 introduced the Analysis Pipeline concept of a statistic and described the settings that statistics share with evaluations. A statistic receives flow records from a filter, computes an aggregate value, and periodically reports that value.

There are two time values that affect statistics: how often to report the statistics, and the length of the time-window used when computing the statistics. The following example reports the statistics every 10 minutes using the last 20 minutes of data to compute the statistic:

```
UPDATE 10 MINUTES
TIME_WINDOW 20 MINUTES
```

- The UPDATE statement specifies the reporting interval; that is, how often to report the statistic. This statement is required in each statistics block.
- The TIME\_WINDOW statement specifies the rolling time frame over which to compute the statistic. When the time window is not specified or specifies a value smaller than the reporting interval, the time window is set to the reporting interval.

Statistics support the aggregation functions (primitives) presented in Section 3.5. Unlike an evaluation, a statistic is simply reporting the function's value, and neither the CHECK statement nor a threshold value are used. Instead, the statistic lists the primitive and any parameters it requires.

Simple examples are:

- Periodically report the number of records:

```
RECORD_COUNT
```

- Periodically report the sum of the packets:

```
SUM PACKETS
```

- Periodically report the average flow duration:

```
AVERAGE DURATION
```

- Periodically report the number of distinct destination ports seen:

```
DISTINCT DPORT
```

- Periodically report the proportion of records for each source port:

```
PROPORTION SPORT
```

## 3.8 List Configuration

Named lists created by internal filters and evaluations can be given extra configuration such that they are responsible for sending updates and alerts independent or in lieu of the mechanism that populates them. If there is a list configuration block, there does not need to be an evaluation block for the configuration file to be accepted. As long as something in pipeline generates alerts, it will run.

Lists created by internal filters have their own timeouts, so they are responsible for removing out-dated elements on their own. Lists populated by evaluations keep track of the timing out of values within the evaluation, and tell the list to remove a value, so those lists know nothing of the timeouts. A result of this is that due to efficiency concerns, some of the alerting functionality described below is not available for lists created and maintained by internal filters. It is explicitly stated which features cannot be used.

This extra configuration is surrounded in a LIST CONFIGURATION block, similar to other pipeline mechanisms. The list to configure must already have been declared before the configuration block.

Section 4.7 shows a list configuration block amidst other pipeline constructions.

Syntax:

```
LIST CONFIGURATION listName
...
END LIST CONFIGURATION
```

The various ways to configure the lists and what alerts can be sent are described below.

### 3.8.1 Alert Triggers

Alerts sent due to a list configuration come from the lists, and have their own timestamps and state kept about their alerts. They are not subject to the alerting restrictions imposed on the evaluations that populate the list.

Lists set up to send alerts must have one of the three configurations listed below.

#### Periodic

The full contents of the list can be packaged into one alert periodically. Syntax:

```
UPDATE timeval
```

This will send out the entire list every 12 hours.

```
UPDATE 12 HOURS
```

#### Element Threshold

An alert can be sent if the number of elements in the list meets a certain threshold, as it's possible that while the contents are important, and can be configured to be sent periodically, knowing the count got above a threshold could be more time sensitive.

Syntax:

```
ALERT MORE THAN elementThreshold ELEMENTS
```

This alert will only be sent the first time the number of elements crosses the threshold. There can also be a reset threshold that if the number of elements drops below this value, pipeline will once again be allowed to send an alert if the number of elements is greater than the alert threshold. There is no alert sent upon going below the reset threshold. The elements in the list are not reset by this either.

Syntax:

```
ALERT MORE THAN elementThreshold ELEMENTS RESET AT resetThreshold
```

This example will send an alert if there are more than 10 elements in the list. No more alerts will be sent unless the number of elements drops below 5, and then it will alert if the number of elements goes above 10 again.

```
ALERT MORE THAN 10 ELEMENTS RESET AT 5
```

The resetting functionality cannot be used by lists created by internal filters.

#### Alert on Removal

Pipeline can send an alert any time a value is removed from the list.

Syntax:

```
ALERT ON REMOVAL
```

Alerting on removal cannot be used by lists created by internal filters.

### 3.8.2 Output Files

In addition to being included in alerts, the contents of the lists can be saved to disk in a specified file. The fields used in the list determine the way the data is saved. With each update sent, this file will be completely overwritten with the current contents of the list. It will NOT be appended. To compute a "diff" of successive files, or to keep any sort of history, post-processing will need to be done outside of Pipeline.

If there is only a single field and it is of type IPV4\_ADDRESS or IPV6\_ADDRESS it will be saved as an IPSet file. Any other type will be saved as a watchlist with one data value per line in ascii format. If there are multiple fields, making a tuple, the data will be saved using the double square bracketed format used by "typeable tuples". For example, if the field list consists of SIP,SPORT, the format of the output file will be:

```
[[1.1.1.1, 80],[2.2.2.2,8080],[3.3.3.3,22]]
```

If the field(s) used has a textual representation that Pipeline can handle, these files can be used as watchlists with (NOT)IN\_LIST in FILTERS.

#### SEED plus OVERWRITE\_ON\_UPDATE

Lists used to hold SIP, DIP, or NHIP can be given a set of initial values by providing an ipset file. Only IPs can be used to seed the lists. Lists containing values of non-IP address can still use SEED plus OVERWRITE\_ON\_UPDATE to generate an output file, but the list will NOT be seeded with the values. In this case, the use of OUTPUT\_FIELD is preferred.

Syntax:

```
SEED pathToSeedFile
```

For reverse compatibility purposes, SEED can be used with non IP lists as a location to write the contents of the list on update, similar to OUTPUT\_FILE above. If used with non-IP values, no true seeding of the list will take place, the filename will be just be used for overwriting if specified. When seeding lists, the seeded values become owned by the lists. Thus if they timeout like other elements would, they will be removed from the list even though they were part of the seed file. Elements used to seed the list have no priority over elements pipeline adds from processing. Elements used to seed the list are not permanent members of the list.

To specify the overwriting of the SEED file on update, use OVERWRITE\_ON\_UPDATE in the LIST CONFIGURATION block.

Syntax:

```
SEED "path/to/seedFile.txt"
OVERWRITE ON UPDATE
```

If not seeding the list, but writing the data to a file on update, use OUTPUT\_FILE as described below instead of SEED combined with OVERWRITE\_ON\_UPDATE.

#### OUTPUT\_FILE

To specify the file name for the list data to be sent to on update, but without seeding the list, use OUTPUT\_FILE. The path to the file can be either local or absolute. The file extension does not matter.

Syntax:

```
OUTPUT FILE "path/to/outputFile.txt"
```

The use of OUTPUT\_FILE implies that Pipeline is to overwrite this file on update, so OVERWRITE\_ON\_UPDATE cannot be used with OUTPUT\_FILE.

OUTPUT\_FILE can be used in concert with SEED. If so, the list will be initialized with values from the SEED file, but the contents of the list will ONLY be written to the OUTPUT\_FILE, and the SEED file will remain unchanged.

### 3.8.3 Other Options

#### Element Threshold to Shutdown

As with evaluations, lists can be configured to shut down if they become filled with too many elements. This is provided as a sanity check to let the user know if the configuration has a flaw in the analysis. If the number of elements meets the shutdown threshold, an alert is sent, the list is freed, and is disconnected from the mechanism that had been populating it.

Syntax:

```
SHUTDOWN MORE THAN shutdownThreshold ELEMENTS
```

Unlike evaluations' shutdown thresholds, shutting down lists like this has no way to restart them after a specified time period.

#### Severity

A severity level must be provided to give context to alerts. It is not used during processing, but included in alerts sent from the lists.

Syntax:

```
SEVERITY integerSeverity
```

## 3.9 List Bundles

Named lists, and ipset files, can now be linked such that if an element is added to all of the lists in the bundle, Pipeline can send an alert, and if desired, add that element to another named list, which can be used in a LIST CONFIGURATION block described above.

The lists referenced in the list bundle must already have been created in the configuration file. All lists must be made up of the exact same fields. An IPSet file can be added to the bundle, provided that the field for the lists is SIP or DIP, and must be put in quotation marks.

High Level Syntax:

```
LIST BUNDLE listBundleName
  existingListNameOrIPSetFilename
  existingListNameOrIPSetFilenameWithSameFields
  ...
  Other options
END LIST BUNDLE
```

### 3.9.1 Named lists for bundle

Each list to be added to the bundle goes on its own line. This list must be created already in the configuration file by an evaluation or internal filter. If this list is to be made from an IPSet, it must be in quotes.

### 3.9.2 Add element to another list

Once an element has been found to be in all of the lists in a bundle, it is then able to be put in a new named list. This list can be used in LIST CONFIGURATION just like any other named list. There is no timeout needed for this, as the element will be removed from this list if it is removed from an element in the bundle.

```
OUTPUT LIST nameOfNewList
```



### 3.9.3 Severity

A severity level must be provided to give context to alerts. It is not used during processing, but included in alerts sent from the lists.

```
SEVERITY integerSeverity
```

### 3.9.4 Do Not Alert

As with evaluations, you can force the list bundle to not alert, as maybe you just want the values that meet the qualifications of the list bundle to be put into another named list (using OUTPUT LIST above), and get alerts of the contents that way. Just add

```
DO NOT ALERT
```

to the list of statements for the list bundle.

### 3.9.5 List Bundle example

Let's say an evaluation creates a list named myServers, and an internal filter creates a list called interestingIPs, and there is an IPSet file is note named notableIPS.set. To include these lists in a bundle, and to put any IP that is in all lists into a new list named reallyImportantIPs, use the following:

```
LIST BUNDLE myExampleBundle
  myServers
  interestingIPs
  "notableIPS.set"
  OUTPUT LIST reallyImportantIPs
  SEVERITY 4
END LIST BUNDLE
```



## Chapter 4

# Example Configurations

This chapter provides examples that configure the Analysis Pipeline for various types of traffic detection and alerting.

### 4.1 Simple filters and evaluations

This section shows simple FILTER and EVALUATION blocks. Many of these provide useful detection, but are simple enough that they do not warrant a separate section.

The “all” filter is a simple filter to pass all flow records collected by SiLK. Use this filter when you want an evaluation to process all records.

```
FILTER all
END FILTER
```

The following filters can be used to match either incoming or outgoing traffic:

```
FILTER incoming-flows
  TYPENAME IN_LIST [in,inweb,inicmp]
END FILTER
FILTER outgoing-flows
  TYPENAME IN_LIST [out,outweb,outicmp]
END FILTER
```

The “udp-traffic” evaluation checks to see if the proportion of UDP flow records is greater than 25% in a 5 minute period.

```
EVALUATION udp-traffic
  FILTER all
  CHECK THRESHOLD
    PROPORTION PROTOCOL 17 >= 25 PERCENT
    TIME_WINDOW 5 MINUTES
  END CHECK
  SEVERITY 5
  ALERT JUST_NEW_THIS_TIME
  ALERT ALWAYS
  CLEAR NEVER
END EVALUATION
```

The following evaluation alerts when an internal host talks to more than 25 different destination ports in a 3 minute window. While that behavior is normal for a server inside the network, it is probably unusual behavior for a individual user's machine inside the network. The FOREACH SIP statement causes the value to be computed for each unique source address. The evaluation uses DISTINCT DPORT to count the number of distinct destination ports, and the evaluation alerts when that count gets above the threshold of 25.

```
EVALUATION too-many-ports
  FILTER outgoing-flows
  FOREACH SIP
  CHECK THRESHOLD
    DISTINCT DPORT > 25
    TIME_WINDOW 180 SECONDS
  END CHECK
  SEVERITY 5
  ALERT JUST_NEW_THIS_TIME
  ALERT ALWAYS
  CLEAR ALWAYS
END EVALUATION
```

This next evaluation provides a complete example of beacon detection. To be considered a beacon, the suspect flow records must occur at least 5 minutes apart, and there must be at least 4 such records. The evaluation alerts once per minute, and only sends the newly found beacons in those alerts.

```
EVALUATION beacon
  FILTER all
  CHECK BEACON
    COUNT 4 CHECK_TOLERANCE 5 PERCENT
    TIME_WINDOW 5 MINUTES
  END CHECK
  CLEAR NEVER
  SEVERITY 3
  ALERT JUST_NEW_THIS_TIME
  ALERT 1 TIMES 60 SECONDS
END EVALUATION
```

The “between-ephemeral” filter matches flow records where both sides of a TCP conversation are using ephemeral ports (ports greater than 1023). While some well known services operate on an ephemeral port (e.g., some web servers run on port 8080), generally the traffic is considered suspect.

```
FILTER between-ephemeral
  PROTOCOL == 6
  SPORT >= 1024
  DPORT >= 1024
END FILTER
```

The “basic-count” evaluation uses the “between-ephemeral” filter, so it is examining traffic that is already considered suspect. The evaluation alerts if the flow record count exceeds 10,000 in any 60 second period. It is set to alert every time the count is over 10,000, and the state is cleared every time an alert it sent.

```
EVALUATION basic-count
  FILTER between-ephemeral
  CHECK THRESHOLD
    RECORD_COUNT > 10000
    TIME_WINDOW 60 SECONDS
```

```

END CHECK
SEVERITY 5
ALERT JUST_NEW_THIS_TIME
ALERT ALWAYS
CLEAR ALWAYS
END EVALUATION

```

The following evaluation uses a 30 minute time window to compute the average number of bytes per flow record for each distinct destination port. The evaluation alerts when that average is greater than 1MB.

```

EVALUATION large-receiving-ports
  FILTER between-ephemeral
  FOREACH DPORT
    CHECK THRESHOLD
      AVERAGE BYTES >= 1000000
      TIME_WINDOW 30 MINUTES
    END CHECK
  SEVERITY 5
  ALERT JUST_NEW_THIS_TIME
  ALERT ALWAYS
  CLEAR NEVER
END EVALUATION

```

## 4.2 Statistics

This section provides examples that use the STATISTIC block to periodically report a value.

All of the example statistics in this section use the following filter, named “allFlows”, which passes all flow records through to the statistics.

```

FILTER allFlows
END FILTER

```

This statistic block simply outputs the number of records seen every 10 minutes. Since the TIME\_WINDOW statement is not provided, the statistic sets its time window to the update time.

```

STATISTIC numRecords
  UPDATE 10 MINUTES
  FILTER allFlows
  RECORD_COUNT
  SEVERITY 1
END STATISTIC

```

The following block is similar to the previous, in that it counts records. However, the addition of the FOREACH statement causes this statistic to count the number of records for each distinct source address. A report is generated every hour.

```

STATISTIC recordsPerSIP
  UPDATE 1 HOUR
  FILTER allFlows
  FOREACH SIP
    RECORD_COUNT
    SEVERITY 1
  END STATISTIC

```

The “numUniqueDPorts” statistic periodically reports the number of distinct destination ports seen in the traffic. It alerts every half hour.

```

STATISTIC numUniqueDPorts
  UPDATE 30 MINUTES
  FILTER allFlows
  DISTINCT DPORT
  SEVERITY 1
END STATISTIC

```

This statistic reports the average number of bytes per record, per destination port. The average is calculated and reported every half hour.

```

STATISTIC averageBytesPerPort
  UPDATE 30 MINUTES
  FILTER allFlows
  FOREACH DPORT
  AVERAGE BYTES
  SEVERITY 1
END STATISTIC

```

The “protocolProportions” statistic below uses a 30 minute time window to calculate the proportion of traffic seen for each IP protocol (TCP, UDP, ICMP, etc), but the data is reported every 10 minutes. Since the time window and update times are different, the reporting behavior follows the pattern given in the following table:

Minute mark	Data sent for minutes
10	0–10
20	0–20
30	0–30
40	10–40
50	20–50

```

STATISTIC protocolProportions
  UPDATE 10 MINUTES
  TIME_WINDOW 30 MINUTES
  FILTER allFlows
  PROPORTION PROTOCOL 6
  SEVERITY 1
END STATISTIC

```

## 4.3 Watchlists

The goal of a watchlist is to detect traffic to and/or from an IP on the watchlist. This is an easy task for the Analysis Pipeline, and checking a flow record against an IP list is stateless, so the memory overhead is small—only what is required to hold the watchlist.

Recall the pipeline works on two levels: **filters** quickly remove irrelevant flow records, and **evaluations** process the relevant ones. In the case of a watchlist, the filters check to see if the source and/or destination address of a flow record is in the watchlist. When a match is found, the record moves to the evaluation stage, but there is no processing required as every flow record that has an IP in the watchlist generates an alert.

The watchlist starts with the set of IP addresses of interest. The easiest way to manage those is normally via a binary SiLK IPset file. SiLK’s `rwsetbuild` tool can be used to create a binary IPset file from a textual list of IP addresses. See the SiLK documentation for details.

To compare an address on a flow record with the IPset, specify which address to compare (SIP for source, DIP for destination, ANY IP for either source or destination, or NHIP for the next hop address), the comparison operator IN\_LIST, and the full path to the IPset file.

pipeline periodically checks the modification time of the IPset files it has loaded. If a newer IPset file replaces an existing file, pipeline will reload the file. If an IPset file is deleted, pipeline continues to use the IPs in the original file.

There are two ways a watchlist can be configured:

1. There can be a single filter and single evaluation for each watchlist. This configuration is straightforward.
2. There can be a separate filter for incoming and outgoing flow records. This configuration is more complex than the first option, but it allows each direction to have its own alert severity and alert label.

The first configuration is explained in detail, then the changes required for the second configuration are described.

For the example below, assume we have two watchlists:

1. *reallyBadList* is defined by the IPset stored in `/var/pipeline/config/reallyBadList.set`
2. *kindaWeirdList* is defined by the IPset stored in `/var/pipeline/config/kindaWeirdList.set`

We create the filters, and specify the full path to each of the IPsets directly in the comparisons. It is important that the full path be used, since pipeline will change directory to `/` when it runs.

To create a filter that checks both the source and destination addresses, use:

```
FILTER reallyBadList
  ANY IP IN_LIST "/var/pipeline/config/reallyBadList.set"
END FILTER

FILTER kindaWeirdList
  ANY IP IN_LIST "/var/pipeline/config/kindaWeirdList.set"
END FILTER
```

Each filter is given a name that matches the IPset. The filters use the IN\_LIST operator to compare the addresses against the IPsets. ANY IP causes the filter to check both the source and destination address against the IPset, and return true if either matches.

Now that the filters are in place to identify all traffic to and from the watchlists, evaluations are needed to send the records that pass the filters to the alerting system. Since every record that passes the filters leads directly to an alert, without further calculation, the EVERYTHING\_PASSES check is used in each evaluation.

When using a single filter for each watchlist, the EVALUATION blocks are:

```
EVALUATION reallyBadList
  FILTER reallyBadList
  CHECK EVERYTHING_PASSES
END CHECK
  SEVERITY 5
  ALERT ALWAYS
END EVALUATION

EVALUATION kindaWeirdList
  FILTER kindaWeirdList
  CHECK EVERYTHING_PASSES
END CHECK
  SEVERITY 2
  ALERT ALWAYS
END EVALUATION
```

There is an evaluation for each filter. Examining the parts of the EVALUATION block:

- The evaluation name is the same as the filter name since they are tied together so tightly. This will not always be the case with other evaluation types.
- The SEVERITY of the “reallyBadList” is set to 5, while that of the “kindaWeirdList” is set to 2 as the name implies it is not as important.
- We always want to send alerts upon discovery of a watchlist hit, so set the alerting frequency to ALWAYS.
- Every record that passes the filter triggers an alert, so we need just one check per evaluation, and that check is the EVERYTHING\_PASSES check.

There are no other parameters to set, as no calculations occur at the evaluation stage.

Since the evaluations use the filters, the blocks must appear in order presented above—FILTER before EVALUATION. There are several ways these blocks can be distributed among configuration files:

1. The blocks may appear in a single file.
2. The blocks for “reallyBadList” could be in one file and those for “kindaWeirdList” in another file, where each file name is the same as the list name. In this case, the main configuration file for pipeline (the one specified to the --configuration-file switch), would need to contain:

```
INCLUDE "reallyBadList.conf"
INCLUDE "kindaWeirdList.conf"
```

3. There can be separate files for each type of block. Assuming the obvious names for the files, pipeline’s main configuration would contain:

```
INCLUDE "filters.conf"
INCLUDE "evals.conf"
```

### 4.3.1 Alternate configuration

As noted above, there can be a separate filter and evaluation reflecting whether the source or destination address matched the watchlist. Assuming pipeline monitors traffic at the border of your organization, and assuming the watchlist contains IPs outside your organization, flow records where the source address matches the watchlist are incoming flows, and records where the destination matches are outgoing flows.

The filters to distinguish whether the source or destination address matches are:

```
FILTER reallyBadList_incoming
  SIP IN_LIST "/var/pipeline/config/reallyBadList.set"
END FILTER

FILTER reallyBadList_outgoing
  DIP IN_LIST "/var/pipeline/config/reallyBadList.set"
END FILTER

FILTER kindaWeirdList_incoming
  SIP IN_LIST "/var/pipeline/config/kindaWeirdList.set"
END FILTER

FILTER kindaWeirdList_outgoing
  DIP IN_LIST "/var/pipeline/config/kindaWeirdList.set"
END FILTER
```



Note that ANY IP in the first example has been changed to either SIP or DIP to match the source or destination addresses, respectively. Also note the name change to the filters.

Once again, we need a separate evaluation for each filter, and the names of the evaluations are the same as the names of the filter:

```
EVALUATION reallyBadList_incoming
  FILTER reallyBadList_incoming
  CHECK EVERYTHING_PASSES
  END CHECK
  SEVERITY 4
  ALERT ALWAYS
END EVALUATION
```

```
EVALUATION reallyBadList_outgoing
  FILTER reallyBadList_outgoing
  CHECK EVERYTHING_PASSES
  END CHECK
  SEVERITY 5
  ALERT ALWAYS
END EVALUATION
```

```
EVALUATION kindaWeirdList_incoming
  FILTER kindaWeirdList_incoming
  CHECK EVERYTHING_PASSES
  END CHECK
  SEVERITY 1
  ALERT ALWAYS
END EVALUATION
```

```
EVALUATION kindaWeirdList_outgoing
  FILTER kindaWeirdList_outgoing
  CHECK EVERYTHING_PASSES
  END CHECK
  SEVERITY 2
  ALERT ALWAYS
END EVALUATION
```

In this example, the severity for outgoing traffic is slightly higher than that for incoming traffic. This reflects that an internal host contacting a host on the watchlist or replying to a query from a watchlist is more severe.

### 4.3.2 DNS Watchlist

A DNS watchlist checks for domain names queried by users in the network that are in a list of known malicious domains. These are different than other watchlists for three reasons. The first is the format of the watchlist which tells Pipeline that the values are to be interpreted as DNS names versus regular strings. The second difference is that the domain names are likely to come from a YAF data source, meaning there could be multiple dnsQNames in each record. The last difference is that if the record is from a YAF data source, the dnsQName is not in the main part of the record, so the evaluation needs to use EXTRA ALERT FIELD to ensure that the dnsQName makes it into the alert.

To tell Pipeline to interpret both the watchlist file's values, and the values from the records as domain names, the first line of the DNS watchlist file must be "##format:dns". Because of this first line, Pipeline will say that "www.cert.org" matches if "cert.org" is in the watchlist. If the field value matches all of the value in the watchlist, even if not all of the value is used, the comparison will return true.

```
Example file: dnsNames.txt
##format:dns
tools.netsa.cert.org
sei.cmu.edu

FILTER importantDomains
    dnsQName IN LIST "dnsNames.txt"
END FILTER

EVALUATION alertOnDomains
    FILTER importantDomains
    SEVERITY 1
    ALERT ALWAYS
    ALERT EVERYTHING
    CHECK EVERYTHING PASSES
    END CHECK
    EXTRA ALERT FIELD dnsQName
END EVALUATION
```

## 4.4 Passive FTP detection

This section shows a complete example of detecting passive FTP. For a description of passive FTP, see Section 3.5.11.

The following filter matches TCP flow records between two ephemeral ports, where the ports are greater than 50000. This traffic could be passive FTP traffic, or it could be something else.

```
FILTER HighPorts
    SPORT >= 50000
    DPORT >= 50000
    PROTOCOL == 6
END FILTER
```

The following `INTERNAL_FILTER` block accepts the flow records matched by the “HighPorts” filter, and stores information about the records in a list called “highPortIPs”. Note that the list is declared as a `HIGH_PORT_LIST` which instructs `pipeline` on how to store and maintain the list. Information is removed from the list after 15 minutes.

```
INTERNAL_FILTER highPortsInternal
    FILTER HighPorts
    HIGH_PORT_LIST highPortIPs 15 MINUTES
END INTERNAL_FILTER
```

The “ftpControl” filter finds flow records on the FTP control channel. This filter passes its records to the “passiveFTP” evaluation below.

```
FILTER ftpControl
    DPORT == 21
    PROTOCOL == 6
END FILTER
```

The “passiveFTP” evaluation brings everything together. When a flow record matches the “HighPorts” filter, the five-tuple (source and destination address, source and destination port, protocol) is stored for 15 minutes. If during that time a flow record arrives that matches the “ftpControl” filter, an event is added to the output list, where it will stay for 60 seconds (the `OUTPUT_TIMEOUT`). If more than one alert potential occurs in a minute, only one alert will be sent.

```

EVALUATION passiveFTP
  INTERNAL_FILTER highPortsInternal
  FILTER ftpControl
  FOREACH SIP DIP SPORT DPORT PROTOCOL
  CHECK HIGH_PORT_CHECK
    LIST highPortIPs
  END CHECK
  SEVERITY 2
  OUTPUT_TIMEOUT 60 SECONDS
  ALERT SINCE_LAST_TIME
  ALERT 1 TIMES 60 SECONDS
  CLEAR ALWAYS
END EVALUATION

```

## 4.5 Web server detection

The example in this section uses a pair of evaluations to identify the web servers on a network, and the destination addresses that are talking to those web servers most often.

The first evaluation, named “findWebServers,” finds the web servers on a network (defined by the “ourNetwork” filter) and adds the web server addresses to the “webServerList” list. The “webServers” filter uses that list to find flow records that represent responses from the web servers. Those response records feed into a second evaluation, named “webVisitors”, which reports the hosts that are visiting the web servers.

To find the servers on a network, look for internal hosts that have many connections with hosts outside the network. The best way to do that is to look at outgoing flow records.

First, define the network to monitor. To monitor the entire network as seen by SiLK, use:

```

FILTER ourEntireNetwork
  TYPENAME IN_LIST [out,outweb]
END FILTER

```

This example monitors a subnet of the monitored network. The filter checks the source address against a subset of the monitored network, and—assume SiLK is monitoring the network’s border—those flows must be outgoing. The filter block is:

```

FILTER ourNetwork
  SIP == 192.168.10.0/24
END FILTER

```

The evaluation that finds the web servers is shown here. An explanation of the evaluation follows.

```

1  EVALUATION findWebServers
2    FILTER ourNetwork
3    FOREACH SIP
4    CHECK THRESHOLD
5      DISTINCT DIP > 25000
6      TIME_WINDOW 600 SECONDS
7    END CHECK
8    CHECK THRESHOLD
9      SUM BYTES > 1000000
10     TIME_WINDOW 600 SECONDS
11    END CHECK

```

```

12 SEVERITY 2
13 OUTPUT_TIMEOUT 1 DAY
14 OUTPUT_LIST SIP webServerList
15 CLEAR ALWAYS
16 ALERT EVERYTHING
17 ALERT 1 TIMES 1 HOUR
18 ALERT ON REMOVAL
19 END EVALUATION

```

This evaluation defines a web server as a source address (FOREACH SIP) that sends data to more than 25,000 distinct destination addresses (line 5) and sends more than 1MB (1,000,000 bytes) of data (line 9), all in a 10 minute window (lines 6 and 10).

The evaluation generates an alert containing the full list of web server addresses (set up using ALERT EVERYTHING) at most once an hour (line 17). It keeps a web server in the output list for up to 1 day (line 13). On line 14, it places each web server address it finds into the list called “webServerList”. If a web server address does not assert itself as a web server every day, the address will be removed from this list.

If a SIP stops acting like a web server for 1 DAY (the length of the OUTPUT TIMEOUT and is removed from the output list, (line 18) will cause pipeline to send out an alert saying that the particular SIP was removed from the list of output entries.

Given that the web servers have been identified, create a filter that matches all flow records coming from the web servers. These flows represent the web server’s responses to requests.

```

FILTER webServers
  SIP IN_LIST webServerList
END FILTER

```

The “webVisitors” evaluation takes the flow records found by the “webServers” filter (line 2), bins the records by the destination address (line 3), and checks to see if any of these addresses have received more than 10MB of data in the last 15 minutes (lines 5–6). However, (line 18) if more than 10000 output entries are active at a given time, shutdown this evaluation (and send out an alert) because that is too many outputs to find and something is wrong with the way the evaluation was set up.

```

1 EVALUATION webVisitors
2   FILTER webServers
3   FOREACH DIP
4     CHECK THRESHOLD
5       SUM BYTES > 10000000
6       TIME_WINDOW 15 MINUTES
7   END CHECK
8   SEVERITY 2
9   ALERT JUST_NEW_THIS_TIME
10  ALERT 1 TIMES 60 SECONDS
11  CLEAR ALWAYS
12  SHUTDOWN MORE THAN 10000 OUTPUTS
13 END EVALUATION

```

## 4.6 IPv6 tunneling detection

Three popular types of IPv6 tunneling are Teredo, 6to4, and ISATAP. The details of each are slightly different, but their general format is the same. IPv6 tunneling happens in two stages:

1. Using IPv4, a node communicates with a well known server (or servers) that provides the initial configuration for the tunnel.

2. A node communicates with another node using IPv6, where the IPv6 traffic is encapsulated in IPv4 and treats the IPv4 Internet as a link layer. The encapsulated traffic has particular port and/or protocol numbers.

Detect any of these IPv6 tunneling mechanisms occurs in three stages:

1. A FILTER block watches for outgoing traffic going to one of the well known initialization servers. An INTERNAL\_FILTER block is used to add the internal hosts that match the filter to a list of “interesting hosts”. Since this list could potentially become very large, hosts are generally removed from list after some period of time.
2. A second FILTER block watches for “interesting hosts” that are also using the the port and/or protocol specified by the particular tunneling mechanism.
3. An EVALUATION block combines the previous two stages. All traffic matching those stage is treated as IPv6 tunneled traffic. Since no other computations are needed in the evaluation, the EVERYTHING\_PASSES primitive is used to send all of the flow records directly into outputs and alerts.

The specifics for detecting each tunneling technique are described in the following sections.

#### 4.6.1 Teredo

The Teredo protocol provides a way to encapsulate IPv6 packets within IPv4 UDP datagrams. A host implementing Teredo can gain IPv6 connectivity with no cooperation from the local network environment.

When a host wishes to initialize a Teredo connection, it first connects to a Teredo server. The first step in detecting Teredo is to find outgoing traffic going to one of the well-known servers.

```
FILTER teredoInitialConnections
# replace IPs with the list of actual Teredo servers
# this could also be done using an IPset
DIP IN_LIST [10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4]
END FILTER
```

Next, take each flow record representing a connection to the Teredo server (i.e., a flow record matched by “teredoInitialConnections”), and store the source address in the list “teredoSIPS”. The addresses are removed from the list after an hour.

```
INTERNAL_FILTER buildTeredoList
  FILTER teredoInitialConnections
  SIP teredoSIPS 1 HOUR
END INTERNAL_FILTER
```

To find the outgoing Teredo traffic, look for flow records on the “teredoSIPS” list that communicate with UDP port 3544:

```
FILTER teredoTraffic
DPORT == 3544
PROTOCOL == 17          # UDP
SIP IN_LIST teredoSIPS
END FILTER
```

The DPORT and PROTOCOL comparisons occur first since they likely to remove many of the flow records from consideration and are faster than the address check.

Lastly, create an evaluation that outputs all records found by the “teredoTraffic” filter to the alerting stage of **pipeline**. This evaluation alerts every time. It has been recommended that the severity level for Teredo be higher than the levels for 6to4 and ISATAP.

```

EVALUATION teredo
  FILTER teredoTraffic
  CHECK EVERYTHING_PASSES
  END CHECK
  CLEAR ALWAYS
  SEVERITY 4
  ALERT SINCE_LAST_TIME
  ALERT ALWAYS
END EVALUATION

```

## 4.6.2 6to4

6to4 is intended as a mechanism to use during the transition from IPv4 to IPv6. In 6to4, internal hosts or subnets communicate with IPv6. To communicate with other hosts or subnets, a 6to4 gateway encapsulates the IPv6 traffic in IPv4 for transmission to a public IPv4 address also acting as a 6to4 gateway where the IPv6 traffic is decapsulated. The IPv4 traffic uses protocol 41.

To find the 6to4 traffic, first look for outgoing traffic destined for a 6to4 server:

```

FILTER 6to4InitialConnections
  # replace 10.0.0.1 with the address of an IPv6 tunnel server
  DIP == 10.0.0.1
END FILTER

```

Take the flow records representing connections to the 6to4 server, and store their internal addresses in the list "6to4SIPS", which has a timeout value of one hour.

```

INTERNAL_FILTER build6to4List
  FILTER 6to4InitialConnections
  SIP 6to4SIPS 1 HOUR
END INTERNAL_FILTER

```

The 6to4 traffic will be traffic from a host on the "6to4SIPS" list that uses protocol 41.

```

FILTER 6to4Traffic
  PROTOCOL == 41          # IPv6 encapsulation
  SIP IN_LIST 6to4SIPS
END FILTER

```

Finally, send all traffic found by the "6to4Traffic" filter to the alerting stage.

```

EVALUATION 6to4
  CLEAR ALWAYS
  SEVERITY 3
  ALERT SINCE_LAST_TIME
  ALERT ALWAYS
  FILTER 6to4Traffic
  CHECK EVERYTHING_PASSES
  END CHECK
END EVALUATION

```

### 4.6.3 ISATAP

ISATAP (Intra-Site Automatic Tunnel Addressing Protocol) is an IPv6 transition mechanism that connects dual-stack (IPv4/IPv6) nodes over an IPv4 network. A host using ISATAP is configured with a potential routers list (PRL) which the host occasionally probes.

Since the initialization servers (the PRL) are fairly dynamic, pipeline should be configured to read the PRL from an IPset. When the IPset file changes, pipeline will notice the change, load the new file, and update the PRL:

```
FILTER isatapConnectPRL
  DIP IN_LIST "/var/pipeline/config/isatapRouters.set"
END FILTER
```

The name `isatapRouters.set` is an example file name, any name will suffice. To update the PRL, overwrite the IPset file with a new file containing the new list. Instead of using an IPset, one could specify the PRL addresses directly in the `FILTER` block (as shown in the Teredo example above), but updating the PRL would require restarting pipeline.

Detecting ISATAP traffic requires finding a host that first connects to a potential router and then begins communicating using protocol 41. To get the lists of IPs that connect to a potential router, create an `INTERNAL_FILTER` as shown here:

```
INTERNAL_FILTER buildIsatapList
  FILTER isatapConnectPRL
  SIP isatapSIPS 1 HOUR
END INTERNAL_FILTER
```

The source addresses for flows that match the “`isatapConnectPRL`” filter are added to the “`isatapSIPS`” list. The hosts on that list are expired after an hour.

Now, take that list of hosts and check for hosts using protocol 41:

```
FILTER isatapTraffic
  PROTOCOL == 41          #IPv6 encapsulation
  SIP IN_LIST isatapSIPS
END FILTER
```

The following evaluation sends flow records matched by the “`isatapTraffic`” filter and passes the flows to the alerting stage of pipeline. The evaluation will alert every time:

```
EVALUATION isatap
  FILTER isatapTraffic
  CHECK EVERYTHING_PASSES
END CHECK
  CLEAR ALWAYS
  SEVERITY 3
  ALERT SINCE_LAST_TIME
  ALERT ALWAYS
END EVALUATION
```

## 4.7 Chaining Lists

Given a watchlist of known malicious IP addresses, look to see if there are IP addresses that many of our nodes communicate with after contacting a malicious IP. This is based on the idea of one IP address infecting nodes, and then another being used for command and control. The time outs and thresholds in this example are not based on any analysis, just made up for simplicity.

This example uses a chain of internal filters to build lists of the different stages, then subsequent filters using this lists. The last piece is an evaluation to see if more than 5 of our IP addresses travel to a specific node after going to an IP on the watchlist.

Look for flows with traffic going to nodes in the watchlist (for simplicity, the assumption is that any traffic going to these nodes comes from inside our network).

```

FILTER UsToBadGuys
    DIP IN_LIST "/ipsets/badGuys.set"
END FILTER

```

For each of the flows going to the watchlist, put the SIP in a list named `ourTalkers`, so now we've stored IPs that have contacted the watchlist. Only keep these IPs in the list for an hour as we're looking for timely transitions to other IPs.

```

INTERNAL_FILTER writeUsDown
    FILTER UsToBadGuys
    SIP ourTalkers 1 HOUR
END INTERNAL_FILTER

```

Now look for flows from our talkers to IP addresses that are not in the original watchlist.

```

FILTER UsToOthers
    SIP IN_LIST ourTalkers
    DIP NOT_IN_LIST "/ipsets/badGuys.set"
END FILTER

```

For each of the flows from our talkers to IP address that are not in the original watchlist, place the DIPs in a list named "placesAfterBG" for an hour. After this, we'll have a list of all of the places our IP addresses sent traffic to after contacting a bad guy(the BG from the list name).

```

INTERNAL_FILTER whereWeGo
    FILTER UsToOthers
    DIP placesAfterBG 1 HOUR
END INTERNAL_FILTER

```

Now that we have the IPs that our nodes contacted after the watchlist, we can identify the traffic from our IPs to these nodes. (This is also done by the filter `UsToOthers`, but we want the other destinations in a list for updates to be sent in the last part of this example).

```

FILTER UsToSecondLevelBG
    SIP IN_LIST ourTalkers
    DIP IN_LIST placesAfterBG
END FILTER

```

We have the traffic we want, so we bin up state by DIP, using `FOREACH DIP`, because we are trying to isolate destination IPs that might be command and control. We use the `distinct` primitive because we want to count the number of unique SIPs that contact each of the DIPs.

```

EVALUATION lotsOfUsThere
    SEVERITY 4
    FILTER UsToSecondLevelBG
    FOREACH DIP
        CHECK THRESHOLD
            DISTINCT SIP > 5
            TIME_WINDOW 1 HOUR
        END CHECK
    END EVALUATION

```



Even though we are sending alerts from the evaluation containing the IPs that more than five of our IPs contact after contacting an IP on the watchlist, we still may want to know all of the places our IPs go recently after contacting a watchlist, so we configure the list created in an earlier internal filter to send its contents every half hour.

```
LIST CONFIGURATION placesAfterBG
  SEVERITY 4
  UPDATE 30 MINUTES
END LIST CONFIGURATION
```



# Appendix A

## Manual Page

This appendix provides the manual page for the **pipeline** daemon. Parts of this manual page are described in more detail in Chapter 2, Installation. The manual page is included here for completeness and easy reference.

### A.1 NAME

**pipeline** - Examine SiLK Flow, YAF, or IPFIX records as they arrive

### A.2 SYNOPSIS

There are 4 possible data sources: SiLK, YAF, IPFIX, or a configuration file with all of the details.

There are 4 possible input modes, 3 of which run continuously and will be run as a daemon by default: UDP or TCP socket (which require **--break-on-recs**), polling a directory for new files. The last is a finite list of files to process, which is never run as a daemon.

Allowable combinations: SiLK with directory polling or named files. YAF with UDP or TCP sockets or named files. IPFIX with UDP or TCP sockets, directory polling, or named files.

A data source configuration file contains all necessary details of both the data source and the input method.

There are 4 general input modes for **pipeline**, each of which can be run with **snarf** and without **snarf**.

To run **pipeline** when built with **snarf**, a snarf destination can be specified with: **--snarf-destination=ENDPOINT**.

To run **pipeline** when built without **snarf**, alert log files must be specified with: **--alert-log-file=FILE\_PATH --aux-alert-file=FILE\_PATH**

In the examples below, substitute the above alerting configurations in place of "ALERT CONFIGURATION OPTIONS".

To run **pipeline** continuously but not as a daemon:

```
pipeline --configuration-file=FILE_PATH
  ALERT CONFIGURATION OPTIONS
  { --silk | --yaf | --ipfix }
  { --udp-port=NUMBER | --tcp-port=NUMBER |
    --incoming-directory=DIR_PATH --error-directory=DIR_PATH
    [--archive-directory=DIR_PATH] [--flat-archive]
  }
  [--break-on-recs=NUMBER]
```

```

{ [--time-is-clock] | [--time-field-name=STRING] |
  [--time-from-schema] |
  [--time-field-ent=NUMBER --time-field-id=NUMBER]
}
[--polling-interval=NUMBER] [--polling-timeout=NUMBER ]
[--country-code-file=FILE_PATH]
[--site-config-file=FILENAME]
--do-not-daemonize

```

To run **pipeline** over a finite list of files:

```

pipeline --configuration-file=FILE_PATH
  ALERT CONFIGURATION OPTIONS
  { --silk | --yaf | --ipfix }
  --name-files
  [--break-on-recs=NUMBER]
  { [--time-is-clock] | [--time-field-name=STRING] |
    [--time-from-schema] |
    [--time-field-ent=NUMBER --time-field-id=NUMBER]
  }
  [--polling-interval=NUMBER] [--polling-timeout=NUMBER ]
  [--country-code-file=FILE_PATH]
  [--site-config-file=FILENAME]

```

To run **pipeline** using a configuration file specifying all data source and data input options. Daemonizing can be turned off if needed.

```

pipeline --configuration-file=FILE_PATH
  ALERT CONFIGURATION OPTIONS
  --data-source-configuration-file=FILE_PATH
  [--country-code-file=FILE_PATH]
  [--site-config-file=FILENAME]
  { --do-not-daemonize |
    { --log-destination=DESTINATION |
      --log-directory=DIR_PATH [--log-basename=BASENAME] |
      --log-pathname=FILE_PATH
    }
    [--log-level=LEVEL] [--log-sysfacility=NUMBER]
    [--pidfile=FILE_PATH]
  }

```

To run **pipeline** continuously as a daemon:

```

pipeline --configuration-file=FILE_PATH
  ALERT CONFIGURATION OPTIONS
  { --silk | --yaf | --ipfix }
  { --udp-port=NUMBER | --tcp-port=NUMBER |
    --incoming-directory=DIR_PATH --error-directory=DIR_PATH
    [--archive-directory=DIR_PATH] [--flat-archive]
  }
  [--break-on-recs=NUMBER]
  { [--time-is-clock] | [--time-field-name=STRING] |
    [--time-from-schema] |
  }

```

```

    [--time-field-ent=NUMBER --time-field-id=NUMBER]
}
[--polling-interval=NUMBER] [--polling-timeout=NUMBER ]
[--country-code-file=FILE_PATH]
[--site-config-file=FILENAME]
{ --log-destination=DESTINATION
  | --log-directory=DIR_PATH [--log-basename=BASENAME]
  | --log-pathname=FILE_PATH
}
[--log-level=LEVEL] [--log-sysfacility=NUMBER]
[--pidfile=FILE_PATH]

```

Help options:

```
pipeline --configuration-file=FILE_PATH --verify-configuration
```

```
pipeline --help
```

```
pipeline --version
```

## A.3 DESCRIPTION

The Analysis Pipeline program, **pipeline**, is designed to be run over three different types of input. The first, as in version 4.x, is files of SiLK Flow records as they are processed by the SiLK packing system. The second type is data coming directly out of **YAF** (or **super\_mediator**) including deep packet inspection information. The last is any raw IPFIX records.

**pipeline** requires a configuration file that specifies *filters* and *evaluations*. The filter blocks determine which flow records are of interest (similar to SiLK's **rwfilter(1)** command). The evaluation blocks can compute aggregate information over the flow records (similar to **rwuniq(1)**) to determine whether the flow records should generate an alert. Information on the syntax of the configuration file is available in the *Analysis Pipeline Handbook*.

The output that **pipeline** produces depends on whether support for the snarf alerting library was compiled into the **pipeline** binary, as described in the next subsections.

Either form of output from **pipeline** includes country code information. To map the IP addresses to country codes, a SiLK prefix map file, *country\_codes.pmap* must be available to **pipeline**. This file can be installed in SiLK's install tree, or its location can be specified with the `SILK_COUNTRY_CODES` environment variable or the **-country-codes-file** command line switch.

### Output Using Snarf

When **pipeline** is built with support for the **snarf** alerting library (<http://tools.netsa.cert.org/snarf/>), the **-snarf-destination** switch can be used to specify where to send the alerts. The parameter to the switch takes the form `tcp://HOST:PORT`, which specifies that a **snarfd** process is running on *HOST* at *PORT*. When **-snarf-destination** is not specified, **pipeline** uses the value in the `SNARF_ALERT_DESTINATION` environment variable. If it is not set, **pipeline** prints the alerts encoded in JSON (JavaScript Object Notation). The outputs go to the log file when running as a daemon, or to the standard output when the **-name-files** switch is specified.

### Legacy Output Not Using Snarf

When **snarf** support is not built into **pipeline**, the output of **pipeline** is a textual file in pipe-delimited (|) format describing which flow records raised an alert and the type of alert that was raised. The location of the output file must be specified

via the **-alert-log-file** switch. The file is in a format that a properly configured ArcSight Log File Flexconnector can use. The *pipeline.sdkfilereader.properties* file in the *share/analysis-pipeline/* directory can be used to configure the ArcSight Flexconnector to process the file.

**pipeline** can provide additional information about the alert in a separate file, called the auxiliary alert file. To use this feature, specify the complete path to the file in the **-aux-alert-file** switch. This option is required.

**pipeline** will assume that both the alert-log-file and the aux-alert-file are under control of the **logrotate(8)** daemon. See the *Analysis Pipeline Handbook* for details.

## Integrating pipeline into the SiLK Packing System

Normally **pipeline** is run as a daemon during SiLK's collection and packing process. **pipeline** runs on the flow records after they have been processed **rwflowpack(8)**, since **pipeline** may need to use the class, type, and sensor data that **rwflowpack** assigns to each flow record.

**pipeline** should get a copy of each incremental file that **rwflowpack** generates. There are three places that **pipeline** can be inserted so it will see every incremental file:

- **rwsender(8)**
- **rwreceiver(8)**
- **rwflowappend(8)**

We describe each of these in turn. If none of these daemons are in use at your site, you must modify how **rwflowpack** runs, which is also described below.

### rwsender

To use **pipeline** with the **rwsender** in SiLK 2.2 or later, specify a **-local-directory** argument to **rwsender**, and have **pipeline** use that directory as its incoming-directory, for example:

```
rwsender ... --local-directory=/var/silk/pipeline/incoming ...

pipeline ... --incoming-directory=/var/silk/pipeline/incoming ...
```

### rwreceiver

When **pipeline** is running on a dedicated machine separate from the machine where **rwflowpack** is running, one can use a dedicated **rwreceiver** to receive the incremental files from an **rwsender** running on the machine where **rwflowpack** is running. In this case, the incoming-directory for **pipeline** will be the destination-directory for **rwreceiver**. For example:

```
rwreceiver ... --destination-dir=/var/silk/pipeline/incoming ...

pipeline ... --incoming-directory=/var/silk/pipeline/incoming ...
```

When **pipeline** is running on a machine where an **rwreceiver** (version 2.2. or newer) is already running, one can specify an additional **-duplicate-destination** directory to **rwreceiver**, and have **pipeline** use that directory as its incoming directory. For example:

```
rwreceiver ... --duplicate-dest=/var/silk/pipeline/incoming ...

pipeline ... --incoming-directory=/var/silk/pipeline/incoming ...
```

## rwflowappend

One way to use **pipeline** with **rwflowappend** is to have **rwflowappend** store incremental files into an archive-directory, and have **pipeline** process those files. However, since **rwflowappend** stores the incremental files in subdirectories under the archive-directory, you must specify a **--post-command** to **rwflowappend** to move (or copy) the files into another directory where **pipeline** can process them. For example:

```
rwflowappend ... --archive-dir=/var/silk/rwflowappend/archive
               --post-command='mv %s /var/silk/pipeline/incoming' ...

pipeline ... --incoming-directory=/var/silk/pipeline/incoming ...
```

**Note:** Newer versions of **rwflowappend** support a **--flat-archive** switch, which places the files into the root of the archive-directory. For this situation, make the archive-directory of **rwflowappend** the incoming-directory of **pipeline**:

```
rwflowappend ... --archive-dir=/var/silk/pipeline/incoming

pipeline ... --incoming-directory=/var/silk/pipeline/incoming ...
```

## rwflowpack only

If none of the above daemons are in use at your site because **rwflowpack** writes files directly into the data repository, you must modify how **rwflowpack** runs so it uses a temporary directory that **rwflowappend** monitors, and you can then insert **pipeline** after **rwflowpack** has processed the files.

Assuming your current configuration for **rwflowpack** is:

```
rwflowpack --sensor-conf=/var/silk/rwflowpack/sensor.conf
           --log-directory=/var/silk/rwflowpack/log
           --root-directory=/data
```

You can modify it as follows:

```
rwflowpack --sensor-conf=/var/silk/rwflowpack/sensor.conf
           --log-directory=/var/silk/rwflowpack/log
           --output-mode=sending
           --incremental-dir=/var/silk/rwflowpack/incremental
           --sender-dir=/var/silk/rwflowappend/incoming

rwflowappend --root-directory=/data
            --log-directory=/var/silk/rwflowappend/log
            --incoming-dir=/var/silk/rwflowappend/incoming
            --error-dir=/var/silk/rwflowappend/error
            --archive-dir=/var/silk/rwflowappend/archive
            --post-command='mv %s /var/silk/pipeline/incoming' ...

pipeline --silk --incoming-directory=/var/silk/pipeline/incoming
        --error-directory=/var/silk/pipeline/error
        --log-directory=/var/silk/pipeline/log
        --configuration-file=/var/silk/pipeline/pipeline.conf
```

## Non-daemon mode

There are two ways to run **pipeline** in non-daemon mode. The first is to run it using one of the ways above that runs forever (socket or directory polling) but just not run it as a daemon. use **-do-not-daemonize** to keep the process in the foreground.

The other way is to run **pipeline** over files whose names are specified on the command line. In this mode, **pipeline** stays in the foreground, processes the files, and exits. None of the files specified on the command line are changed in any way—they are neither moved nor deleted. To run **pipeline** in this mode, specify the **-name-files** switch and the names of the files to process.

## A.4 OPTIONS

Option names may be abbreviated if the abbreviation is unique or is an exact match for an option. A parameter to an option may be specified as **-arg=param** or **-arg param**, though the first form is required for options that take optional parameters.

### General Configuration

These switches affect general configuration of **pipeline**. The first two switches are required:

#### **-configuration-file=FILE\_PATH**

Give the path to the configuration file that specifies the *filters* that determine which flow records are of interest and the *evaluations* that signify when an alert is to be raised. This switch is required.

#### **-country-codes-file=FILE\_PATH**

Use the designated country code prefix mapping file instead of the default.

#### **-site-config-file=FILENAME**

Read the SiLK site configuration from the named file *FILENAME*. When this switch is not provided, the location specified by the `SILK_CONFIG_FILE` environment variable is used if that variable is not empty. The value of `SILK_CONFIG_FILE` should include the name of the file. Otherwise, the application looks for a file named *silk.conf* in the following directories: the directories `$SILK_PATH/share/silk/` and `$SILK_PATH/share/`; and the *share/silk/* and *share/* directories parallel to the application's directory.

#### **-dns-public-suffix-file=FILENAME**

**pipeline** comes with a public suffix file provided by Mozilla at: [https://publicsuffix.org/list/public\\_suffix\\_list.dat](https://publicsuffix.org/list/public_suffix_list.dat). To provide **pipeline** with a different list, use this option to provide a file. The file must be formatted the same way as Mozilla's file. This is optional.

#### **-stats-log-interval=NUMBER**

The number of integer minutes between **pipeline** logging statistics regarding records processed and memory usage. Setting this value to 0 turns off this feature. This is optional and the default value is 5 minutes.

### Data Source Configuration Options

**pipeline** needs to know what general type of data it will be receiving, SiLK flows, YAF data, or raw IPFIX. If there are multiple data sources, a data source configuration file is required. If using a daemon config file, the data source configuration file variable is required.

If there is a single data source, the data source type can be specified on the command line. Depending on the type of data, there are different available options for receiving data.



**-silk**

The records are SiLK flows. The data input method options are the same as in past versions **-incoming-directory=DIR\_PATH** Pipeline will poll a directory forever for new flow files **-name-files** The list of files pipeline will process are listed on the command line as the last group of arguments

**-yaf**

The records are coming directly from a **YAF** sensor (or from an instance of **super\_mediator**). The data input options are: **-udp-port=NUMBER** and **-break-on-recs=NUMBER** UDP socket to listen for YAF data on, and how many records to process before breaking and running evaluations. **-tcp-port=NUMBER** and **-break-on-recs=NUMBER** TCP socket to listen for YAF data on, and how many records to process before breaking and running evaluations. **-name-files** Process YAF data files listed on the command line.

**-ipfix**

The records are raw IPFIX records, not coming directly from YAF. The data input options are: **-udp-port=NUMBER** and **-break-on-recs=NUMBER** UDP socket to listen for YAF data on, and how many records to process before breaking and running evaluations. **-tcp-port=NUMBER** and **-break-on-recs=NUMBER** TCP socket to listen for YAF data on, and how many records to process before breaking and running evaluations. **-name-files** Process YAF data files listed on the command line. **-incoming-directory=DIR\_PATH** Pipeline will poll a directory forever for new flow files

**-data-source-configuration-file=FILENAME**

The data source and input options are detailed in a configuration file. The syntax for the file can be referenced by the Pipeline Handbook.

## Timing Source Configuration Options

If the primary (or only) data source is SiLK, these options are not used. If it is a SiLK data source, flow end time is still used for timing source.

Otherwise, one of these options is required to provide a timing source.

**-time-is-clock**

Use the system clock time as the timing source

**-time-field-name=STRING**

Use the provided field name as the timing source.

**-time-field-ent=NUMBER and -time-field-id=NUMBER**

These must be used together, as it takes an enterprise ID and an element ID to define an information element. This element will be used as the timing source.

**-time-from-schema**

Use the timing source specified by the schema. If no timing source is specified by the schema(s) used, **pipeline** will report an error.

**-break-on-recs=NUMBER**

Versions 4.x only worked on SiLK files, which provided an easy way to know when to stop processing/filtering records and run evaluations. When accepting a stream of records from a socket, there is no break, so **pipeline** needs to know how many records to process/filter before running evaluations. Use this option to tell **pipeline** how many records to process. This option is required for socket connections.

## Alert Destination when Snarf is Available

When **pipeline** is built with support for snarf (<http://tools.netsa.cert.org/snarf/>), the following switch is available. Its use is optional.

### **-snarf-destination=ENDPOINT**

Specify where **pipeline** is to send alerts. The *ENDPOINT* has the form `tcp://HOST:PORT`, which specifies that a **snarfd** process is running on *HOST* at *PORT*. When this switch is not specified, **pipeline** uses the value in the `SNARF_ALERT_DESTINATION` environment variable. If that variable is not set, **pipeline** prints the alerts locally, either to the log file (when running as a daemon), or to the standard output.

## Alert Destination when Snarf is Unavailable

When **pipeline** is built without support for snarf, the following switches are available, and the **-alert-log-file** switch is required.

### **-alert-log-file=FILE\_PATH**

Specify the path to the file where **pipeline** will write the alert records. The full path to the log file must be specified. **pipeline** assumes that this file will be under control of the **logrotate(8)** command.

### **-aux-alert-file=FILE\_PATH**

Have **pipeline** provide additional information about an alert to *FILE\_PATH*. When a record causes an alert, **pipeline** writes the record in textual format to the **alert-log-file**. Often there is additional information associated with an alert that cannot be captured in a single record; this is especially true for statistic-type alerts. The **aux-alert-file** is a location for **pipeline** to write that additional information. The *FILE\_PATH* must be an absolute path, and **pipeline** assumes that this file will be under control of the **logrotate(8)** command.

## Daemon Mode

The following switches are used when **pipeline** is run as a daemon. They may not be mixed with the switches related to Processing Existing Files described below. The first two switches are required, and at least one switch related to logging is required.

### **-incoming-directory=DIR\_PATH**

Watch this directory for new SiLK Flow files that are to be processed by **pipeline**. **pipeline** ignores any files in this directory whose names begin with a dot (.). In addition, new files will only be considered when their size is constant for one polling-interval after they are first noticed.

### **-polling-interval=NUMBER**

Sets the interval in seconds for how often **pipeline** checks for new files if polling a directory using **-incoming-directory**

### **-polling-timeout=NUMBER**

Sets the amount of time in seconds **pipeline** will wait for a new file when polling a directory using **-incoming-directory**

### **-udp-port=NUMBER**

Listen on a UDP port for YAF or IPFIX records, not SiLK records. **pipeline** will reestablish this connection if the sender closes the socket, unless **-do-not-reestablish** is used.

### **-tcp-port=NUMBER**

Listen on a TCP port for YAF or IPFIX records, not SiLK records. **pipeline** will reestablish this connection if the sender closes the socket, unless **-do-not-reestablish** is used.

### **-error-directory=DIR\_PATH**

Store in this directory SiLK files that were NOT successfully processed by **pipeline**.

One of the following mutually-exclusive logging-related switches is required:

**-log-destination=DESTINATION**

Specify the destination where logging messages are written. When *DESTINATION* begins with a slash /, it is treated as a file system path and all log messages are written to that file; there is no log rotation. When *DESTINATION* does not begin with /, it must be one of the following strings:

**none**

Messages are not written anywhere.

**stdout**

Messages are written to the standard output.

**stderr**

Messages are written to the standard error.

**syslog**

Messages are written using the **syslog(3)** facility.

**both**

Messages are written to the syslog facility and to the standard error (this option is not available on all platforms).

**-log-directory=DIR\_PATH**

Use *DIR\_PATH* as the directory where the log files are written. *DIR\_PATH* must be a complete directory path. The log files have the form

*DIR\_PATH/LOG\_BASENAME-YYYYMMDD.log*

where *YYYYMMDD* is the current date and *LOG\_BASENAME* is the application name or the value passed to the **-log-basename** switch when provided. The log files will be rotated: at midnight local time a new log will be opened and the previous day's log file will be compressed using **gzip(1)**. (Old log files are not removed by **pipeline**; the administrator should use another tool to remove them.) When this switch is provided, a process-ID file (PID) will also be written in this directory unless the **-pidfile** switch is provided.

**-log-pathname=FILE\_PATH**

Use *FILE\_PATH* as the complete path to the log file. The log file will not be rotated.

The following switches are optional:

**-archive-directory=DIR\_PATH**

Move incoming SiLK Flow files that **pipeline** processes successfully into the directory *DIR\_PATH*. *DIR\_PATH* must be a complete directory path. When this switch is not provided, the SiLK Flow files are deleted once they have been successfully processed. When the **-flat-archive** switch is also provided, incoming files are moved into the top of *DIR\_PATH*; when **-flat-archive** is not given, each file is moved to a subdirectory based on the current local time: *DIR\_PATH/YEAR/MONTH/DAY/HOUR/*. Removing files from the archive-directory is not the job of **pipeline**; the system administrator should implement a separate process to clean this directory.

**-flat-archive**

When archiving incoming SiLK Flow files via the **-archive-directory** switch, move the files into the top of the archive-directory, not into subdirectories of the archive-directory. This switch has no effect if **-archive-directory** is not also specified. This switch can be used to allow another process to watch for new files appearing in the archive-directory.

**-polling-interval=NUM**

Configure **pipeline** to check the incoming directory for new files every *NUM* seconds. The default polling interval is 15 seconds.

**-log-level=LEVEL**

Set the severity of messages that will be logged. The levels from most severe to least are: **emerg**, **alert**, **crit**, **err**, **warning**, **notice**, **info**, **debug**. The default is **info**.

**-log-sysfacility=NUMBER**

Set the facility that **syslog(3)** uses for logging messages. This switch takes a number as an argument. The default is a value that corresponds to **LOG\_USER** on the system where **pipeline** is running. This switch produces an error unless **-log-destination=syslog** is specified.

**-log-basename=LOG\_BASENAME**

Use **LOG\_BASENAME** in place of the application name for the files in the log directory. See the description of the **-log-directory** switch.

**-pidfile=FILE\_PATH**

Set the complete path to the file in which **pipeline** writes its process ID (PID) when it is running as a daemon. No PID file is written when **-do-not-daemonize** is given. When this switch is not present, no PID file is written unless the **-log-directory** switch is specified, in which case the PID is written to *LOGPATH/pipeline.pid*.

**-do-not-daemonize**

Force **pipeline** to stay in the foreground—it does not become a daemon. Useful for debugging.

## Process Existing Files

**-name-files**

Cause **pipeline** to run its analysis over a specific set of files named on the command line. Once **pipeline** has processed those files, it exits. This switch cannot be mixed with the Daemon Mode and Logging and Daemon Configuration switches described above. When using files named on the command line, **pipeline** will not move or delete the files.

## Help Options

**-verify-configuration**

Verify that the syntax of the configuration file is correct and then exit **pipeline**. If the file is incorrect or if it does not define any evaluations, an error message is printed and **pipeline** exits abnormally. If the file is correct, **pipeline** simply exits with status 0.

**-print-schema-info**

Print the information elements available based on the schemas that arrive. When using any data source other than SiLK flows, this feature requires data to arrive such that templates/schemas can be read and information elements made available. This option will not verify your configuration file.

**-show-schema-and-verify**

Print the information elements available based on the schemas that arrive, and verify the syntax of the configuration file. When using any data source other than SiLK flows, this feature requires data to arrive such that templates/schemas can be read and information elements made available.

**-help**

Print the available options and exit.

**-version**

Print the version number and information about how the SiLK library used by **pipeline** was configured, then exit the application.

## A.5 ENVIRONMENT

### SILK\_CONFIG\_FILE

This environment variable is used as the value for the **–site-config-file** when that switch is not provided.

### SILK\_COUNTRY\_CODES

This environment variable allows the user to specify the country code mapping file that **pipeline** will use. The value may be a complete path or a file relative to the SILK\_PATH. If the variable is not specified, the code looks for a file named *country\_codes.pmap* in the location specified by SILK\_PATH.

### SILK\_PATH

This environment variable gives the root of the install tree. As part of its search for the SiLK site configuration file, **pipeline** checks for a file named *silk.conf* in the directories *\$ SILK\_PATH/share/silk* and *\$ SILK\_PATH/share*. To find the country code prefix map file, **pipeline** checks those same directories for a file named *country\_codes.pmap*.

### SNARF\_ALERT\_DESTINATION

When **pipeline** is built with snarf support ( <http://tools.netsa.cert.org/snarf/>), this environment variable specifies the location to send the alerts. The **–snarf-destination** switch has precedence over this variable.

## A.6 SEE ALSO

**silk(7)**, **rwflowappend(8)**, **rwflowpack(8)**, **rwreceiver(8)**, **rwsender(8)**, **rwfilter(1)**, **rwuniq(1)**, **syslog(3)**, **logrotate(8)**, <http://tools.netsa.cert.org/snarf>, *Analysis Pipeline Handbook*, *The SiLK Installation Handbook*