AirCERT: The Definitive Guide

Brian Trammell

Roman Danyliw

Sean Levy

Andrew Kompanek

AirCERT: The Definitive Guide
by Brian Trammell, Roman Danyliw, Sean Levy, and Andrew Kompanek

Table of Contents

List of Tables

List of Figures

Chapter 1. Introducing AirCERT

1.1. What is AirCERT?

Automated Incident Reporting (AirCERT) is an Internet-scalable distributed system for sharing security event data among administrative domains. Using AirCERT, organizations can exchange security data ranging from raw intrusion alerts generated automatically by network intrusion detection systems and related sensor technology, to network flow data, to incident reports created and maintained by human analysts. The infrastructure is designed around several standard formats for exchanging reports, including IODEF, SNML and IDMEF, and provides a set of configurable data normalization tools for adapting data from a broad range of security technologies to the AirCERT framework.[1] This framework automates the process of sanitization, normalization, and sharing -- enabling cooperation and coordination on an otherwise impractical scale, and making possible a whole new class of analyses.[2]

The goal of AirCERT is to provide a capability to discern trends and patterns of intruder activity spanning multiple administrative domains. The underlying assumption is that given a sample of data from representative sites, it is possible to draw these conclusions; and with a larger enough sample, extrapolate activity at different sites. With regard to the collected data, the premise to AirCERT is that the sum is more than the individual parts.

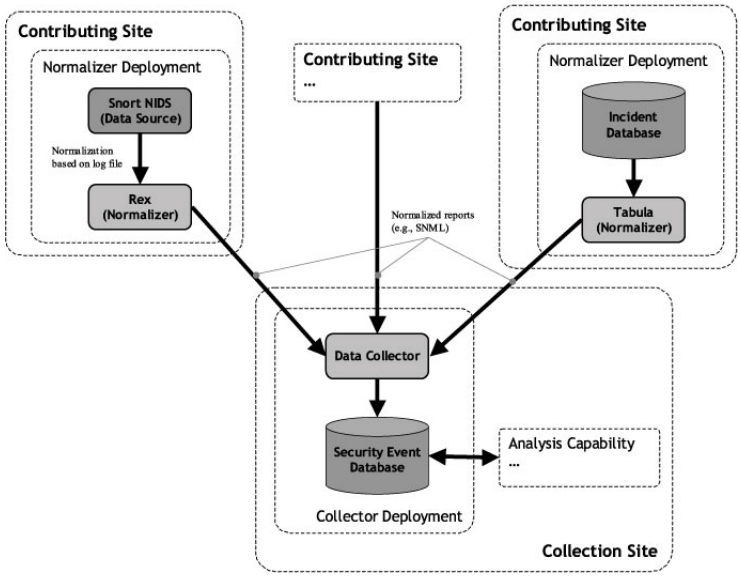
The analytical products of AirCERT will enhance an organization's security posture by providing a framework to automatically confirm activity that the local security infrastructure detected, as well as, provide trends occurring on the larger Internet that may impact the organization.

1.2. Conceptual Architecture

The AirCERT system is based on a distributed architecture in which a site opt share or receive with any number of other sites. Deployments are based on *Normalizers* that extract and normalize data from local security event data sources (e.g., IDS or firewall), sanitize that data, and feed it to *Collectors* which coalesce and store this security data in a format suitable for analysis. Associated with these collectors is an analytical capability which further processes the collected data, as well as a *Publisher* that is responsible for exchanging the data with other collectors. [3] Arbitrary topologies are possible to mimic the complexities of organizational and personal trust relationships. Hence, the exact topology of a given AirCERT deployment will depend on the goals and policies of the cooperating organizations.[4]

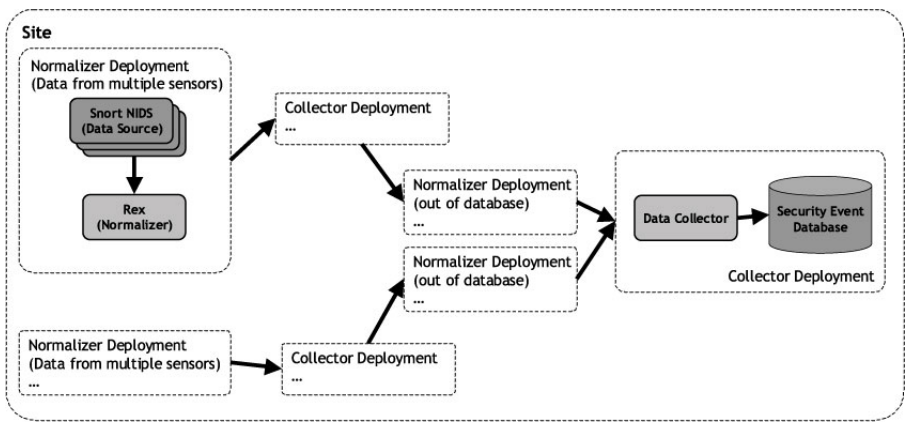
The deployment in Figure 1-1 is based on a single primary Collector populating a database. Several Normalizers feed the Collector data, derived from several different types of sources. This arrangement supports an outsourced aggregated analysis capability, in which several organizations agree to share data with a trusted analysis center.

Figure 1-1. Abstract Architecture of centralized AirCERT Deployment



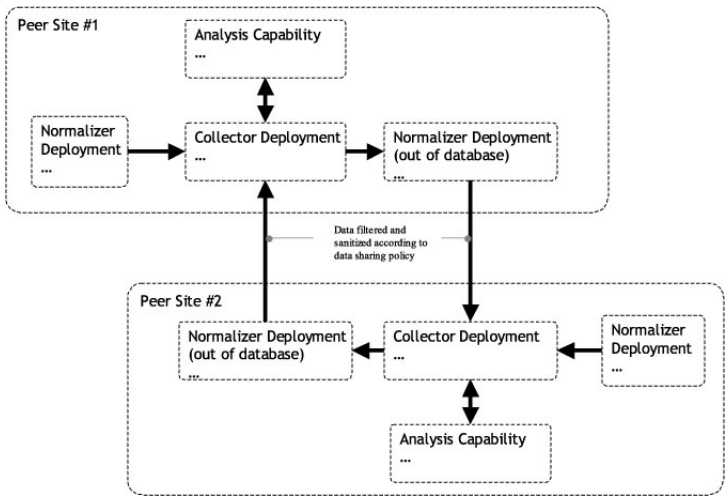
A given site site may also have its own data collection capability, with its own collection hierarchy. Figure 1-2, illustrates a site that maintains a central database which aggregates data collected from different local databases. The primary database is populated by Normalizers which retrieve from local incident databases (which themselves were populated by Normalizers).

Figure 1-2. A More Complex Site Deployment



Both of the above examples illustrate tree-like topologies. However, the AirCERT architecture does not prevent sites from exchanging data in both directions. A site that collects incident data might also *publish* that data to other collectors, by configuring normalizers that that retrieve data from a Collector's database, allowing for any number of different configurations. A simple peer-oriented topology is shown in Figure 1-3.

Figure 1-3. A Peer-Oriented Topology. Arrows illustrate data flow.



1.2.1. Normalization Architecture

The normalization platform is responsible for normalizing and transmitting relevant security event data from a technology-specific data store to the centralized data store of the collection platform. Normalization involves converting a proprietary data representation to a standardized format used in AirCERT. This transformation requires both reformatting of the data (e.g., field 1 in format 1 is now field 3 in format 2), as well as, semantic translations (e.g., signature 6 from IDS 1 and signature 8 from IDS 2 really detected the same event, therefore, in AirCERT, it shall be known as signature 10).

Given that almost all security technologies export data in a different format, it was not scalable to write a script or program for each technology. Rather, the AirCERT approach is to segment the technologies according to the data store or transmission protocol used to log events. A quick survey revealed that the vast majority of existing tools are built upon text files, RDBMS, fixed-record binary files, Event log, or SNMP. Hence, AirCERT initially targeted writing normalizers for text files (i.e., rex) and RDBMS (i.e., tabula). The philosophy with implementing a given normalizer is to provide the primitives to extract the data and support a descriptive language to describe the format of this data. Given a way to describe the format of the data, a mapping to an AirCERT data representation is possible.

After the normalization of the data is complete, the remaining element of the normalization platform is the transmission engine (i.e., dredge) that sends normalized data to the collection platform. The normalization and transmission process were explicitly decoupled in order to allow graceful handling of transient network errors and optimize the use of scarce local site resources (e.g., bandwidth).

Figure 1-4. Normalization Architecture.

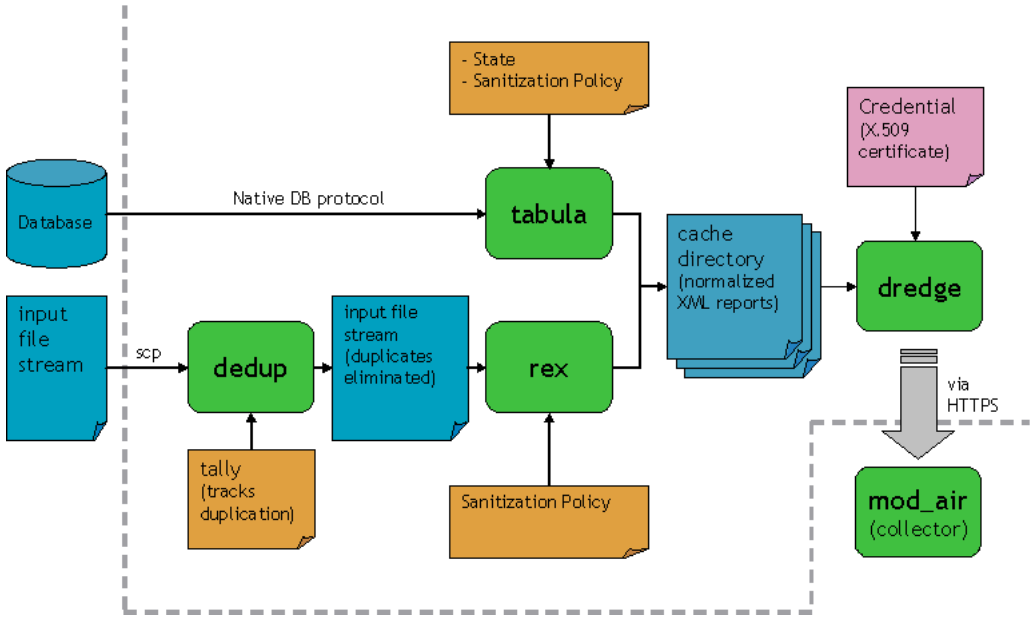


Figure 1-4 illustrates the the data flow between all the components of the normalization platform. Initially, data is extracted from its original data store. In the case of a RDBMS, tabula, the database normalizer, uses the native DB protocol (or ODBC) to connect to the database and extract all the new events of interest per some previously initialized state and configuration. With a text file, dedup, a text-file pre-parser, is first invoked. This utility reads through the log file to identify the first new unprocessed record and provides this information to rex, the text-file normalizer. This two-step process is required in environments where log rotation and normalization occur in isolation.

After the each of the normalizers parses the data, it is transformed into an AirCERT format (i.e., an XML document) and written to a cache directory (explicitly in the file-system). In typical configurations, each event corresponds to a single file, however, optimizations are possible.

Running in tandem to the normalization process is dredge, the retransmission engine, that watches for new data in the cache directory. Events from the cache directory are transmitted to the collection platform over HTTPS. Authentication between dredge and the collection platform occurs via X.509 certificates (through TLS). In the case of network latency or failure, data remains spooled in the cache directory.

1.2.2. Collection Architecture

The Collection Platform serves as a repository of a given data set. It provides a mechanism to accept to new data (i.e., aggregate other data set that have been shared) and house this information in a database through a Collector. However, the collection platform also has a Publisher, a capability to export and share information with other collection platforms. Also associated with each repository is an analytical capability to process and distill information from all the collected data stored locally. Just as in the case of data produced by normalizers, these analytical results can be shared.

Figure 1-5. Collection Architecture.

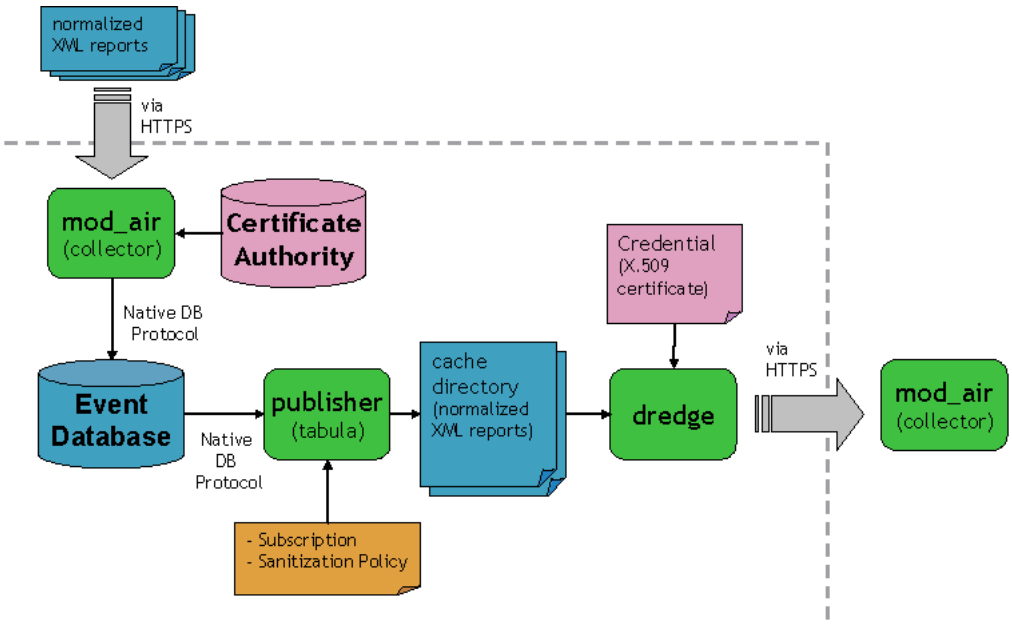


Figure 1-5 illustrates the data flow through the collection platform. Initially, normalized data arrives at the collector over HTTPS. The collector authenticates this data based on the presented certificate (and presumably the client also validates the credentials of the collector prior to sending the data). Using the certificate authority, the credentials are verified to ensure that the client is authorized to communicate with this collector. After, the authentication and authorization phase, the data is processed and stored in a database.

A publisher monitors the contents of the database and identifies data that can be shared with others. When such data is identified, it is converted to a standardized AirCERT XML representation and written to a cache directory. The cache directory is monitored by dredge, that will transmit the data to the appropriate remote collector.

1.3. Communication

Due to the potential sensitivity of the information in question, and the myriad of technologies that might be involved in the AirCERT framework, there are many concerns related to the confidentiality and integrity of the data; and the authentication and authorization of the end-points. Inside an administrative domain, especially with regard to the normalizer platform interacting with the local security infrastructure, best practices and existing mechanisms can be used. For example, securing the connection between the database normalizer and the database itself, can be accomplish through the standard protection mechanism provided by the database. It is with the data sharing and subsequent aggregate storage that requires careful design.

A custom protocol built onto HTTPS is used when exchanging information across administrative domains between contributing and collecting sites. [5] HTTPS is the HTTP protocol tunneled in an SSL/TLS con-

nection. The security properties provided by TLS are used to guarantee the confidentially and integrity of AirCERT data while it is transmitted.

Each end-point (e.g., normalizer, collector, or publisher) in the TLS communication is authenticated with an X.509 certificate. The collection platform has a Certificate Authority (CA) that issues client certificates for all normalizer deployments that will be contributing data to it, and server certificates for each of its collector deployments.

From a practical point of view, this means that in order for a normalizer (through dredge, the transmission engine) to establish a connection with a collector, the following must be present in the normalizer deployment:

- A *CA certificate*, identifying the CA that issued the collector's server certificate and the normalizer client certificate.
- The *server certificate* belonging to the collector to which the normalizer will be sending reports.
- The *client certificate* belonging to this normalizer, issued by the CA.
- The *client key*, the secret key associated with the client certificate.

If a normalizer or publisher exchange information with more than one collection platform, separate credentials (i.e., certificates) will be required.

Figure 1-6. Authentication Domains

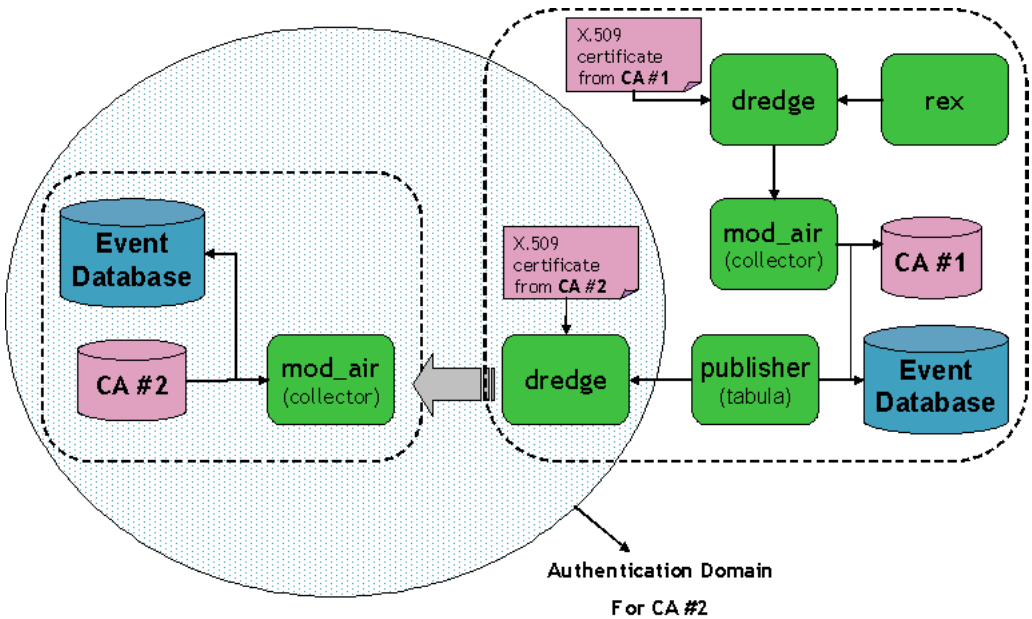


Figure 1-6 demonstrates the relationship between the the collection and normalization components with regard to authentication. A certificate from CA #1 is required to communicate with collector #1, and the same is true for collector #2. Normalizers and publishers have certificates issued by the collection platform with which they communicate.

The use of a public-key infrastructure (PKI) always introduces its own level of complexity. Hence, specific design choices were made to mesh a PKI authentication model into AirCERT. The implication of these choices are a a web-of-trust relationship between collection sites, rather than a strict hierarchy.

- There are no certificate hierarchies. Each collector has a self-signed root CA from which certificates are issued to normalizers and publishers that share data with it.
- A Certificate Revocation List (CRL) is only maintained at the collector (and it is not disseminated to the other participants). The collector uses this internal CRL and other policy information to enforce proper authorization. No security guarantees are provided if the certificates are used between other components.

Typically, these certificates would be obtained through the administrative contact for the collector. Additional details on issuing and installing certificates may be found in the documentation for the various AirCERT components.

1.4. Implementation Philosophy

The applications and libraries that make up the AirCERT framework are based on readily available technologies, that are typically free and released under open source licenses (i.e., GPL and LGPL). Most of the framework is implemented in portable 'C' and the configuration and deployment scripts support several flavors of Unix as well Macintosh OS X.

Notes

1. The current release of AirCERT is based on SNML, although the framework will soon support arbitrary schema for representing reports. No standard format for flow data interchange is yet supported; IPFIX support is slated for a future release.
2. The current release of AirCERT is less concerned with sanitization than centralization of data; only tabula, which is no longer in production use, supports data sanitization. Future releases based on in-progress event transcoder technology will support much more powerful summarization and sanitization on the data source side.
3. Ultimately, the the goal is to involve the architecture into a publish/subscribe system that will allow data to be accessed as needed, rather than relying on copying data between databases
4. Again, this release is focused more on the efficient operation of a distributed network of sensors reporting back to a single central collector.
5. The use of the Intrusion Detection Exchange Protocol (IDXP) might need to be revisited again.

Chapter 2. Workflow

2.1. Introduction

The workflow implemented by an AirCERT deployment may be understood as the five stage process of capturing network data from the wire, generating event data from the packets at the sensor side into alerts or flows, normalizing those into reports or flow records in a standard interchange format, transmitting them to a central collector, and inserting event records into the collector's security event database.

The AirCERT workflow as supported by this release includes all five of these stages; as AirCERT sensors diversify to include deployments within organizations already doing capture and sensor-side processing, four- and three-stage abbreviations of this process will also be supported out of the box.

2.2. Workflow Principles

AirCERT processes are all built around a general assembly-line design pattern. When run in daemon mode, as they are in production use, each process watches for input matching a given file glob expression, then routes successfully processed files and output to another directory, where they will match the input glob expression of subsequent stages.

Synchronization among stages is achieved using file locking; each file that is in use by a process has an associated lockfile. Processes do not release lockfiles until they are done processing, and where possible, no changes are committed until the final step before lockfile deletion. This provides consistent semantics for cleanup - lockfiles present when the system is not running can be safely deleted.

This assembly line pattern presently uses files on disk; this has the advantage of being easy to recover from in case of unexpected shutdown, and the disadvantage of binding the performance of the system as a whole to disk performance. Future revisions of the workflow may use other assembly line storage techniques as available and appropriate, e.g., a shared-memory blackboard.

2.3. AirCERT Workflow Stages

Figure 2-1. Workflow stages and processes on a typical AirCERT sensor.

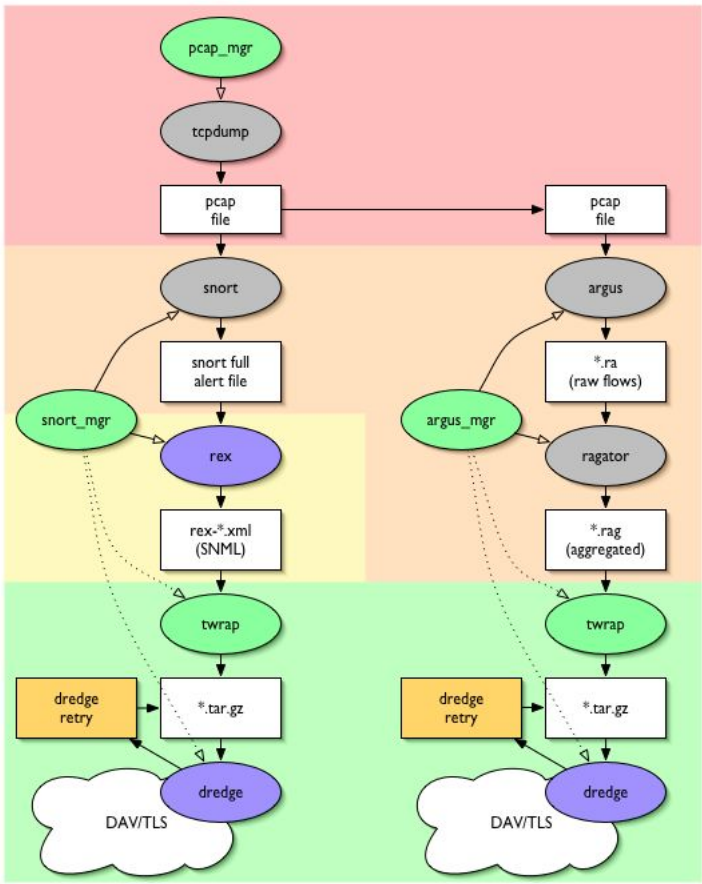
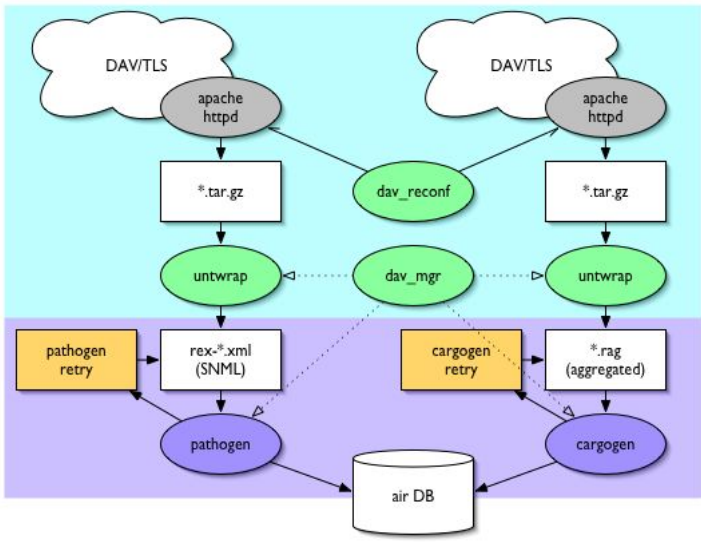


Figure 2-2. Workflow stages and processes on a typical AirCERT collector.



2.3.1. Capture

AirCERT sensors use *tcpdump* to capture full packet data in libpcap format from the sensing interface. The libpcap format was chosen because it is supported by the majority of processors used and considered for future use. Future AirCERT sensors that provide for external capture may use any source for packet data provided that is can be transcoded into this format, and that sufficient packet information is available for the processing stage.

In Figure 2-1, the Capture stage is colored red.

2.3.2. Generation

Raw packet data needs to be processed into event data for analysis. The two sensor-side event data generators currently supported by AirCERT are *snort* (for generation of IDS alert events) and *argus* (for generation of flows).

In Figure 2-1, the Generation stage is colored orange.

2.3.3. Normalization

For data sources generating data which is covered by a supported standardized information exchange format, the normalization stage prepares data for transmission. Two normalization tools are provided with the current distribution of AirCERT, *rex*, a tool for extracting line-oriented event data from a flat file based on regular expressions, and *tabula*, a SQL-based tool for extracting data from a relational database. Only the former is currently maintained in production use, as we do not presently use any sensor-side generator that natively supports RDBMS tuple generation.

Note also that there is no normalization step for Argus data, as raw Argus binary flow data is transmitted to the collector for processing. Future AirCERT releases will use IPFIX to transmit flow data.

In Figure 2-1, the Normalization stage is colored yellow.

2.3.4. Transmission

IDS alert reports and flow data are transmitted to the collector via WebDAV over TLS as compressed POSIX TAR archives. Two processes cooperate to achieve this on the sensor side, *twrap*, an assembly-line daemon for building files into tarballs, and *dredge*, a WebDAV client which supports client certificate authentication and cheap lockfile support.[1]

In Figure 2-1, the Normalization stage is colored green.

2.3.5. Collection

On the other side of the WebDAV connection from Dredge sits the collector. The collector is made up of three components: Apache *mod_dav* provides the WebDAV server, *untwrap* starts the collector-side assembly line by unwrapping tarballs created by twrap, and the collector processors, which place received event data into a relational database.

The *pathogen* collector processor inserts data from arbitrary XML documents into the database; it is used to process rex-normalized alert data from snort. The *cargogen* collector processor inserts raw argus flow data into the database; it is used to process data from argus sensors.

In Figure 2-2, the Collection stage receiver is colored blue, and the Collection stage processor is colored violet.

2.4. Analysis

The relational database populated by the collector is the primary data source for analysis. This AirCERT release also contains a tool for the generation of periodic time-series graphical analysis products, called *CAVE* (for Cheap AirCERT Visualization Environment). Future releases will focus on more featureful analysis and the automated sharing and distribution of analysis instructions and results.

Notes

1. Yes, WebDAV provides a fairly rich set of locking semantics, but actually interfacing to the backend on Apache mod_dav is unsupported an fraught with peril.

Chapter 3. Text File Normalizer (rex)

3.1. Introduction

Rex is the AirCERT text file normalizer, an application which transforms incident data in arbitrary ASCII text file formats (e.g. IDS alert logs, syslog entries) into incident reports in one of three XML interchange formats. Rex ships with a collection of configuration files for common input formats, and provides a rich configuration language for handling formats not supported by the included configurations.

3.2. Installation and Setup

3.2.1. Prerequisites

Rex requires the AirCERT common library (libair), version X.Y.Z or later. libair is available from http://aircert.sourceforge.net/libair, or in the `lib` directory of the AirCERT full source distribution.

3.2.2. Building Rex

As with all AirCERT applications, Rex's build system is autoconf-based; therefore, `./configure ; make ; sudo make install` should work to build and install Pathogen. Rex's build system is tested on Fedora Core (Linux), OpenBSD, and Mac OS X.

3.2.3. Configuring Rex

Rex takes the name of an XML configuration file as a required command-line argument on each invocation. This configuration file binds regular expressions used to extract information from each record in the input file to XML elements and attributes in each output report. Rex ships with a collection of configuration files for common input file formats. These configuration files are installed into *`prefix`*`/etc` when Rex is installed.

Table 3-1. Configuration files supplied with Rex

Filename	Input file type	Output DTD
`rexcfg-snort-full-snml.xml`	Snort full alert log file	SNML v0.3

If the required input and output formats are listed in the table above, simply use the corresponding configuration file when you run Rex. Otherwise, see the manual chapter entitled "The Rex Configuration Language" to write a new Rex configuration file.

3.3. Running Rex

On each invocation, Rex reads a single configuration file and a single input file or stream, and writes a collection of output report files to an output cache directory. While Rex normally terminates on a processing error (on the theory that the input data is correct, and that a Rex configuration file that causes such an error is incomplete), Rex may optionally be made to resynchronize on an input stream after an error, and to drop clippings from the input stream causing an error into an optional fail destination. Rex takes arguments to specify the input file, the output cache directory, and the configuration file, as well as a set of optional arguments to control error recovery and logging output, as follows:

rex { `--config` *`config-file`* } { `--in` *`input-file`* } { `--cachedir` *`cache-directory`* } [`--faildir` *`failure-directory`*] [`--force`] [`--loglevel` *`level`*] [`--stderr`] [`--console`] [`--logfile` *`log-file`*] [`--syslog` *`log-facility`*] [`--ident` *`log-identity`*] [`--help`] [`--version`] [`--copyright`]

Table 3-2. Rex options

Option	Argument	Description
`--config`	configuration file	Rex configuration file to use for normalization
`--in`	input file	file to normalize
`--cachedir`	cache directory	directory to write reports to
`--force`	none	resynchronize after processing error
`--faildir`	directory	destinations for portions of output stream near processing error

As Rex runs, it scans the input file in order, and writes a collection of XML incident reports into the cache directory. Each file in the cache directory will be named `rex-`*`pid-rulename-serial`*`.xml`, where *`pid`* is the process ID of the Rex process that created the file, *`rulename`* is the name of the rule that created the file (from the Rex configuration file), and *`serial`* is a serial number, counting from 0.

By default, Rex produces no logging output. The following options control how Rex will log its operation:

Table 3-3. Rex logging options

Option	Argument	Description
`--loglevel`	one of panic, critical, error (default), alert, warning, notice, info, debug	log level (less to more verbose)
`--stderr`	none	log to standard error stream
`--console`	none	log to system console
`--logfile`	log file name	log to specified file
`--syslog`	syslog facility name	log to the specified facility using syslog(3)

Option	Argument	Description
`--ident`	syslog identity (default rex)	set logging identity

After a successful run of Rex, the reports in the cache directory may be sent to an AirCERT collector using the Dredge report transmitter.

3.4. Running Dedup

Dedup is a record duplicate elimination utility, included with Rex, designed to facilitate the usage of Rex to normalize independently rotated log files. Dedup runs over a collection of input files, and filters out any records that it has already passed, tracking those records in a tally file. When used as a filter in front of Rex, this allows the normalizer to be run multiple times over an input file that may or may not have been truncated (as during log file rotation), and normalize each record only once.

Dedup takes six command line arguments, four of which are required:

dedup { `--filetype` *`filetype-name`* } { `--in` *`input-file`* } { `--out` *`output-file`* } { `--tally` *`tally-file`* } [`--overwrite` | `--append`]

The *`filetype-name`* tells Dedup what type of records are in the input file; this is necessary for Dedup to select the proper driver needed to interpret each record during duplicate detection. The filetypes presently supported by Dedup are listed in the table below.

Table 3-4. File types supported by dedup

Filetype name	Description
`snort-full`	Snort full alert log file

The other three required arguments specify the input, output, and tally files. The special input file - instructs Dedup to read from standard input, and the special output file - instructs Dedup to write to standard output. This latter option is useful when chaining Dedup and Rex directly together using a pipe, as in: **`dedup --filetype my-file-type --tally my.tally --in my.log --out - | rex --in - --config my-rex-config.xml --cachedir cache`**.

If the output file already exists, one of `--overwrite` or `--append` are required, as the default behavior is to require the output file not to exist before writing to it.

The tally file is used to record which records have already been passed across subsequent runs of Dedup. Its format is specific to each filetype. If the specified tally file does not exist, it will be created.

3.5. How Rex Works

Since the Rex configuration file maps in a fairly straightforward way on to the internal data structures used by Rex during normalization, it is useful to first understand how Rex normalizes files in order to understand the structure of the configuration file.

Rex roughly consists of an "input side" and an "output side". The input side is a line-oriented parser

which attempts to match each expression in the configuration file in order to each input line. The line-oriented nature of the input side means that it's necessary to use one of the multiline constructs in the Rex configuration language (delimiters or identifiers) in order to create reports built from data in more than one line in the input file.

The output side consists of a table of active reports (i.e. those which are still being built) and the functions that build those reports. An active report consists of a tree, which will eventually become the output XML document, and a variable table which may be used for scratch space while the report is being built. In most cases, especially in heterogeneous input files, there will be only one active report at any given time. Active reports are keyed by rule name (the `name` attribute of each `<rexrule>` element) and by an identifier, if present. When an active report is made complete, its tree is checked to ensure it is valid for its DTD, then emitted as an XML file into the cache directory. Each output file is named according to the process ID of the Rex process which created it, a serial number (unique within a given run of Rex), and the name of the `<rexrule>` element which created the report.

Rex processes the input file one line at a time. For each line, it attempts to match each expression in each rule in the configuration file, in the order in which they appear. When an expression matches a line in the input file, the input side makes a call to the output side to create a new report if no active report exists for the given rule and optional identifier, then iterates over each of the actions bound to the matched expression. These actions build reports from chunks of data extracted from the input by the matched expression. In this way, Rex reduces the text file normalization problem to one of shuffling small chunks of data from an input record into the attributes and values of XML elements in the output file, optionally rewriting them along the way.

3.6. Configuration Basics

3.6.1. Overview

A Rex configuration file is an XML document which describes to Rex how to normalize a single kind of input file into a single output DTD. This file consists of an options section followed by a set of normalization rules (`<rexrule>` elements). Each normalization rule represents a different kind of report to generate, and consists of a list of Perl 5 compatible regular expressions (`<expression>` elements) to match against the input. Rex's regular expression handling is provided by PCRE (the Perl Compatible Regular Expression library); see its documentation for details on regular expression syntax appropriate for use with Rex. Each `<expression>` contains a number of actions (`<skip>`, `<abort>`, and `<put>`) to be executed when the expression matches.

Each `<rexrule>` may also contain `<put>` actions; these actions are run when each instance of a report described by the rule is first created. See the Actions section below for more on actions.

Figure 3-1. Simple example Rex configuration file

```
<aircert-config>
  <rex>
    <options>
        <option name="dtd" value="snml"/>
    </options>
```

```
    <rexrule name="tcp-packets">
      <put src="'rex" dst="/SNML-Message/sensor/file"/>
      <put src="'4" dst="/SNML-Message/event/packet/iphdr@ver"/>
      <expression pattern="^src (\d+\.\d+\.\d+\.\d+):(\d+)"
                  delimiter="start"
                  shortcircuit="yes">
        <put src="$1"
             dst="/SNML-Message/event/packet/iphdr@saddr"/>
        <put src="$2"
             dst="/SNML-Message/event/packet/iphdr@sport"/>
      </expression>
      <expression pattern="^dst (\d+\.\d+\.\d+\.\d+):(\d+)"
                  shortcircuit="yes">
        <put src="$1"
             dst="/SNML-Message/event/packet/iphdr@daddr"/>
        <put src="$2"
             dst="/SNML-Message/event/packet/iphdr@dport"/>
      </expression>
    </rexrule>
  </rex>
</aircert-config>
```

3.6.2. Housekeeping

The top-level element in a Rex configuration file is `<aircert-config>`. This is a generic top-level element used by all XML-based AirCERT configuration languages. This must in turn contain a single `<rex>` element, which identifies the contents as a Rex configuration.

The `<rex>` element consists of an `<options>` elements containing one or more `<option>` elements, each of which sets the Rex configuration option named by the `name` attribute to the value contained in the `value` element. Rex presently understands two options. The `dtd` option is required; it must presently name a libih-supported report DTD: `snml`, `idmef`, or `iodef`. The `inputfile` option names the path of the input file to normalize. This is useful for cases where the input file's path is always well known and implied by its type. It is optional, and is only honored if the `--in` command-line option is not present.

3.6.3. Actions

As has already been mentioned, Rex works by binding actions to expressions and rules, and runs these actions when those expressions match and reports are created according to those rules. Rex presently supports three actions. `<skip>` is used to conditionally skip processing of an expression based upon data matched by the expression or in the report. `<abort>` is used to conditionally stop processing an active report based upon data matched by the expression or in the report. `<put>` is used to copy data among the matched expression, the report variable table, and the report tree. `<put>` may also optionally transform data from one format into another (controlled via the `xform` attribute).

The `<skip>` and `<abort>` elements each have three attributes: `condition`, `a`, and `b`. `condition` specifies some test involving references `a` and `b` that causes the action to be run if the test is true. As some tests require only one reference, the `b` attribute is optional.

The `<put>` element also has three attributes: `src`, `dst`, and the optional `xform`. `<put>` copies data from the `src` reference into the `dst` reference, optionally applying the transformation described in `xform` along the way.

3.6.4. Reference Notation

Actions use references to address the various data sources and destinations they operate on. A reference is a string that names some chunk of data in a report tree, a report variable table, a captured expression, or in the configuration file itself. The first character of the reference defines what type of data the reference names.

3.6.4.1. Report Tree Paths

If a reference begins with a forward-slash (`/`), it names an IH path in the output report. IH paths are a way of naming each element in the output XML document. Their notation is quite similar to XPath: each element is addressed by the element names from the root of the document up to that element, separated by forward-slashes (`/`). Tree path references may be used as source or destination. Multiple elements of the same kind may be handled in a number of ways:

Table 3-5. Methods of distinguishing multiple elements in an IH path

element(x)	the xth element with the given name (counting from 1)
element(n)	the last element with the given name. This is the most recently added element, which is usually what you want when writing to a newly created multiple element in Rex.
element:attribute=value	the element with the given name whose given attribute has the given value

3.6.4.2. Report Variable Names

If a reference begins with a dollar-sign (`$`) followed by an identifier beginning with an alphabetic character, it names a variable in the output report's variable table. Variable references may be used as source or destination.

3.6.4.3. Captured Subexpressions

If a reference begins with a dollar-sign (`$`) followed by a number, it refers to a captured subexpression resulting from a match of an input line against the expression the action is bound to. Captured subexpres-

sions count leftmost parentheses starting with 1; the special reference `$0` names the entire match. Captured subexpression references may only be used as source, and may only be used within an expression.

3.6.4.4. Literals

If a reference begins with a single quote (`'`), the entire text of the reference after the single quote is treated as a literal string. Literals may only be used as source.

3.7. Types of Rules

3.7.1. One-liners

The simplest rex rule is the "one-liner", a rule that consists of a single expression which matches a single line in the input file. In order to use a one-liner, all of the information required for a report must appear in a single line of the input file.

A one-liner consists of a `<rexrule>` element containing zero or more `<put>` actions and exactly one `<expression>`. An example one-liner and its input and output are shown below.

Figure 3-2. Example of a one-liner rule

```
<rexrule name="one-liner-IP">
  <expression pattern="(\d+\.\d+\.\d+\.\d+) -> (\d+\.\d+\.\d+\.\d+)">
    <put src="$1"
         path="/SNML-Message/event/packet/iphdr@saddr"/>
    <put src="$2"
         path="/SNML-Message/event/packet/iphdr@daddr"/>
  </expression>
</rexrule>
```

Figure 3-3. One-liner example input and output

```
Input:
10.7.11.9 -> 10.7.11.15

Output (rex-[pid]-00000000-one-liner-IP.xml):
<SNML-Message>
  <event>
    <packet>
      <iphdr saddr="10.7.11.9" daddr="10.7.11.15"/>
    <packet>
  </event>
```

```
</SNML-Message>
```

3.7.2. Delimited Reports

When the information required to build one report is spread across multiple lines, it is necessary to tell Rex where one report ends and another begins. The mechanism for doing this is the `delimiter` attribute of the `<expression>` element. This attribute, if present, must be either `start` or `end`. The rules for delimiters are as follows:

1. If a start delimiter expression matches, a new report is created for the given rule. If there is already an active report for the given rule, it is closed and emitted.
2. If an end delimiter expression matches, the current active report is closed and emitted.
3. If there is no active report for a given rule which has a start delimiter, and an expression within that rule that is not a start delimiter matches, the match is ignored.

An example delimited report rule and its input and output are shown below.

Figure 3-4. Example of a delimited report rule

```
<rexrule name="delimited-IP">
  <expression pattern="^src (\d+\.\d+\.\d+\.\d+)"
              delimiter="start">
    <put src="$1"
         dst="/SNML-Message/event/packet/iphdr@saddr"/>
  </expression>
  <expression pattern="^dst (\d+\.\d+\.\d+\.\d+)"
    <put src="$1"
         dst="/SNML-Message/event/packet/iphdr@daddr"/>
  </expression>
</rexrule>
```

Figure 3-5. Example delimited report input and output

```
Input:
src 10.7.11.9
dst 10.7.11.15
src 10.7.11.15
dst 10.7.11.22

Output (rex-[pid]-00000000-delimited-IP.xml):
<SNML-Message>
  <event>
    <packet>
```

```
      <iphdr saddr="10.7.11.9" daddr="10.7.11.15"/>
    <packet>
  </event>
</SNML-Message>

Output (rex-[pid]-00000001-delimited-IP.xml):
<SNML-Message>
  <event>
    <packet>
      <iphdr saddr="10.7.11.15" daddr="10.7.11.22"/>
    <packet>
  </event>
</SNML-Message>
```

3.7.3. Identified Reports

When the information required to build one report is spread across multiple lines and information from multiple different reports is interleaved in the same file, as is the case in certain syslog files, Rex uses "identifying subexpressions" to differentiate between active reports for the same rule. This does require that each line bound to a given report have a string identifier that identifies a given line as belonging to a given report. For syslog files, this could be the process ID of the reporting process; for sendmail logs, this could be the message ID.

If any expression within a rule has an identifying subexpression, all expressions must have one. The identifying subexpression is selected by the `identifier` attribute on the `<expression>` element, and counts parentheses from the left just as captured subexpression references do. If a rule with identifying subexpressions does not have start or end delimiter expressions, it should have a `horizon` attribute. The horizon is an estimate of the maximum number of lines not associated with a given report that can appear between lines associated with that report. It is used to keep the reports table from growing indefinitely, so that Rex may determine that it is safe to close, validate, and emit a report without an end delimiter.

An example identified report rule and its input and output are shown below.

Figure 3-6. Example of an identified report rule

```
<rexrule name="identified-IP" horizon="10">
  <expression pattern="^(\S+) src (\d+\.\d+\.\d+\.\d+)"
              identifier="1">
    <put src="$1"
         dst="/SNML-Message/sensor/file"/>
    <put src="$2"
         dst="/SNML-Message/event/packet/iphdr@saddr"/>
  </expression>
  <expression pattern="^(\S+) dst (\d+\.\d+\.\d+\.\d+)"
              identifier="1">
    <put src="$2"
         dst="/SNML-Message/event/packet/iphdr@daddr"/>
```

```
  </expression>
</rexrule>
```

Figure 3-7. Identified report example input and output

```
Input:
rep1 src 10.7.11.9
rep2 src 10.7.11.15
rep1 dst 10.7.11.15
rep2 dst 10.7.11.22

Output (rex-[pid]-00000000-identified-IP.xml):
<SNML-Message>
  <sensor>
    <file>rep1</file>
  </sensor>
  <event>
    <packet>
      <iphdr saddr="10.7.11.9" daddr="10.7.11.15"/>
    <packet>
  </event>
</SNML-Message>

Output (rex-[pid]-00000001-identified-IP.xml):
<SNML-Message>
  <sensor>
    <file>rep2</file>
  </sensor>
  <event>
    <packet>
      <iphdr saddr="10.7.11.15" daddr="10.7.11.22"/>
    <packet>
  </event>
</SNML-Message>
```

3.8. Advanced Features

3.8.1. Multiple Elements in Reports

By default, if a given path in a given report is written multiple times, the latest write wins. If a report may have multiple instances of one of its elements (e.g. multiple cross-references for a given alert), Rex must be told to create a new element in the report tree at the appropriate time. This is done by setting the

`create` attribute of the first `<put>` action associated with the given multiple element to `yes`. When this `<put>` action is executed, it will first check the destination path (in the `dst` attribute) to see if it is already set, and if so, it will create a new element. Subsequent `<put>` actions to the other attributes of the newly created element should use the `element(n)` notation (see the section above on Report Tree Paths for more) to ensure that data is written to attributes of the most recently created element of the given name.

Note that the `create` attribute of `<put` requires that the destination be a report path, and can only be used from within an `<expression>` element. An example of a report rule handling multiple outputs along with input and output is shown below.

Figure 3-8. Example of a rule handling multiple elements

```
<rexrule name="multi-IP">
  <expression pattern="(\d+\.\d+\.\d+\.\d+) -> (\d+\.\d+\.\d+\.\d+)"
              delimiter="start">
    <put src="$1"
         dst="/SNML-Message/event/packet/iphdr@saddr"/>
    <put src="$2"
         dst="/SNML-Message/event/packet/iphdr@daddr"/>
  </expression>
  <expression pattern="(url) (http.*?)">
    <put src="$1"
         dst="/SNML-Message/event/reference@system"
         create="yes"/>
    <put src="$2"
         dst="/SNML-Message/event/reference(n)"/>
  </expression>
</rexrule>
```

Figure 3-9. Multiple element example input and output

```
Input:
10.7.11.9 -> 10.7.11.15
url http://www.cert.org/bad-things
url http://cve.mitre.org/very-bad-things

Output (rex-[pid]-00000000-multi-IP.xml):
<SNML-Message>
  <event>
    <packet>
      <iphdr saddr="10.7.11.9" daddr="10.7.11.15"/>
    <packet>
    <reference system="url">http://www.cert.org/bad-things</reference>
    <reference system="url">http://cve.mitre.org/worse-things</reference>
  </event>
</SNML-Message>
\end{verbatim}
```

3.8.2. Controlling the Parser

Three attributes on the <expression> element can be used to control the behavior of the regular expression parser: caseless, multimatch and shortcircuit.

By default, the regular expression parser honors case on all regular expressions. To override this behavior, set the caseless attribute to yes.

By default, the regular expression parser stops at the first match for a given expression on each line. When your expression matches a line fragment, and it is acceptable to match that expression multiple times, set the multimatch attribute to yes. Be sure to override the default multiple-write behavior as outlined in the section entitled Multiple Elements in Reports, above.

By default, every single expression in the config file is checked against each input line, in the order in which the expressions appear in the configuration file. It is often desirable to cause a match against one of those expressions to cause the parser to skip the subsequent expressions in the input file and to the next input line. If a match against an expression implies that no other expression will match, skipping over the other expressions can increase performance. If a given line may match two expressions in the configuration file, and you only want one of them to match, it's necessary to skip over the other. The shortcircuit attribute provides this behavior. If it is set to yes on an expression, any match on that expression will cause the parser to ignore the rest of the configuration file for the current input line and skip to the next one. To increase performance, place the expressions which are likely to match in the input file most often toward the beginning of the configuration file, and expressions likely to match infrequently toward the end, and set the shortcircuit attribute. It may also be desirable to place a "trap" expression (which will match against common input lines which won't match any other expression in the file) with the shortcircuit attribute set and no captured subexpressions into the configuration file before a long run of infrequently matched expressions.

3.8.3. Report Variables

In addition to the report tree, which is addressed by paths and is written out as the output XML report, each report has associated with it a variable table addressed by name which may be used as scratch space. This is used to allow information retrieved from a previous match to be used in <skip> and <abort> action conditions, or to conditionally <put> data into different paths in the output report based upon a later match. Take the example of the transport-layer port in an IP address: if the protocol of the IP datagram is not known at the time the port is parsed, it cannot be associated with the appropriate type of header (TCP or UDP) until the transport layer header is parsed. This example is presented below.

Figure 3-10. Example of a rule using report variables

```
<rexrule name="variable-IP">
  <expression pattern="(\d+\.\d+\.\d+\.\d+):(\d+) -> (\d+\.\d+\.\d+\.\d+):(\d+)"
              delimiter="start">
    <put src="$1"
         dst="/SNML-Message/event/packet/iphdr@saddr"/>
```

```
    <put src="$2"
         dst="$sport"/>
    <put src="$3"
         dst="/SNML-Message/event/packet/iphdr@daddr"/>
    <put src="$4"
         dst="$dport"/>
  </expression>
  <expression pattern="TCP">
    <put src="$sport"
         dst="/SNML-Message/event/packet/iphdr/tcphdr@sport"/>
    <put src="$dport"
         dst="/SNML-Message/event/packet/iphdr/tcphdr@dport"/>
    <put src="’6"
         dst="/SNML-Message/ecent/packet/iphdr@proto"/>
  </expression>
  <expression pattern="UDP">
    <put src="$sport"
         dst="/SNML-Message/event/packet/iphdr/udphdr@sport"/>
    <put src="$dport"
         dst="/SNML-Message/event/packet/iphdr/udphdr@dport"/>
    <put src="’17"
         dst="/SNML-Message/ecent/packet/iphdr@proto"/>
  </expression>
</rexrule>
```

Figure 3-11. Report variable example input and output

```
10.7.11.9:3030 -> 10.7.11.15:80
is a TCP packet
10.7.11.15:53 -> 10.7.11.22:53
is a UDP packet

Output (rex-[pid]-00000000-variable-IP.xml):
<SNML-Message>
  <event>
    <packet>
      <iphdr saddr="10.7.11.9" daddr="10.7.11.15" proto="6">
        <tcphdr sport="3030" dport="80"/>
      </iphdr>
    <packet>
  </event>
</SNML-Message>

Output (rex-[pid]-00000001-variable-IP.xml):
<SNML-Message>
  <event>
    <packet>
      <iphdr saddr="10.7.11.15" daddr="10.7.11.22" proto="17">
        <udphdr sport="53" dport="53"/>
```

```
      </iphdr>
    <packet>
  </event>
</SNML-Message>
```

3.8.4. Data Transformation

In all the examples to this point, Rex merely reorganizes small chunks of data; extracting them from the input using regular expressions, and placing them in the output document addressed by IH paths. In some cases, it is necessary to transform these small chunks of data themselves, for example, to translate a timestamp from the format given in the input to the one required by the output, or to translate a hostname into an IP address via a DNS lookup. This functionality is provided by the xform module of the AirCERT utility library libairutil, and is controlled by the optional `xform` attribute on each `<put>` action.

xform is a typed data transformation facility. Each chunk of data handled by xform is tagged with a type. An xform type is a triplet of a semantic type (what the data means), a format type (how the data is represented), and a storage type (how the data is stored in memory -- i.e. the C type of the data).

xform types are named by stringing the semantic, format, and storage type names together separated by dots. If the final dot and the storage type are omitted, they are assumed to be `string` -- this is the only xform storage type supported by Rex. A list of xform data types supported by Rex is listed the table below.

Transformations are described in terms of source and destination types -- the source type tags the incoming data (i.e. as it exists in the input file), and the destination type describes the data as it should appear in the output. These "type twins" are named by joining the source and destination types with a colon. If a transformation does not change the semantic type of the data (as most do not), the destination semantic type (but not its trailing dot) may be omitted from the type twin. For example, the type twin `host.name.string:host.dotquad.string` could be expressed as `host.name:.dotquad`.

To transform data before placing it in the output report, place the type twin naming the desired transformation in the `xform` attribute of the appropriate `<put>` action.

Table 3-6. xform data types supported by Rex

Type	Description
`any.string`	String data
`any.string-upper`	String data, uppercased
`any.string-lower`	String data, lowercased
`any.string-hex`	String data, hex encoded
`host.name`	Internet host FQDN
`host.dotquat`	Internet (v4) host address in dotted-quad format
`host.name`	Internet host address in integer format
`ip-service.name`	UDP or TCP service name
`ip-service.int`	UDP or TCP service well-known destination port
`ip-protocol.name`	IP protocol name

Type	Description
`ip-protocol.int`	IP protocol number
`time.snort-year`	Snort full alert timestamp with year
`time.snort-noyear`	Snort full alert timestamp without year
`time.iso8601`	ISO 8601 timestamp
`ip-flags.snort`	Snort full alert IP fragmentation flags
`ip-flags.int`	IP fragmentation flags, reserved MSB
`tcp-flags.snort`	Snort full alert TCP flags
`tcp-flags.int`	TCP flags, reserved MSB
`ip-option.snort`	Snort format IP option data
`ip-option.net-hex`	Network-order hexadecimal IP option data
`tcp-option.snort`	Snort default format TCP option data
`tcp-option.snort-sack`	Snort SAck TCP option data
`tcp-option.snort-ts`	Snort timestamp TCP option data
`tcp-option.snort-wscale`	Snort window scale TCP option data
`tcp-option.snort-mss`	Snort MSS TCP option data
`tcp-option.snort-cc`	Snort T/TCP option data
`tcp-option.net-hex`	Network-order hexadecimal TCP option data

3.9. Configuration File Reference

3.9.1. `<aircert-config>`

Common top-level element in all AirCERT configuration files.

Parent	None; top-level element
Children	• 1 `<rex>`
Attributes	• `author` (optional): the author of this configuration file. • `date` (optional): the revision date of this configuration file. • `name` (optional): the name of this configuration file. • `version` (optional): the version number of this configuration file.

3.9.2. `<rex>`

Top-level element for Rex-specific configuration elements.

Parent	<aircert-config>
Children	• 1 <options> • 1..n <rexrule>
Attributes	None

3.9.3. <options>

Container for a list of <option>s.

Parent	<rex>
Children	• 0..n <option>
Attributes	None

3.9.4. <option>

Sets a Rex global configuration option. Two options are presently supported:

- dtd (required): output DTD; one of snml, idmef, or iodef.
- inputfile (optional): the input file to read; this can be overridden by the --in command-line option.

Parent	<options>
Children	None
Attributes	• name (required): the name of the option to set. • value (required): the value to set it to.

3.9.5. <rexrule>

Defines a Rex normalization rule. Each rule builds a specific type of report. For processing heterogeneous input files, only one rule per configuration file is required. A rule is made up of a list of <put> actions to execute when a new report is created, followed by a list of <expression>s to match against each line of the input file.

Parent	<rex>
Children	• 0..n <put> • 1..n <expression>

Attributes	• `name` (required): the name of the rule. This name will appear in the filename of the output reports. • `horizon` (optional): the number of lines in the input file without a match on an identified report before that report can be considered complete.

3.9.6. `<expression>`

Defines a regular expression to match against each line of an input file and the actions to execute on each match.

Parent	`<rexrule>`
Children	• 0..n `<skip>` • 0..n `<abort>` • 0..n `<put>`
Attributes	• `pattern` (required): the Perl 5 compatible regular expression to match against the input. • `delimiter` (optional): `start` if this expression is a report start delimiter, `end` if this expression is a report end delimiter. • `identifier` (optional): The captured subexpression number of this pattern's identifier if this expression is part of an identified report. • `shortcircuit` (optional): `yes` to skip to the next input line when this expression matches. The default behavior is to attempt to match every expression in the configuration file in order against each input line. • `multimatch` (optional): `yes` to attempt to match this expression multiple times against each input line. The default behavior is to move on to the next expression in the input file after the first match of an expression against the input line. • `caseless` (optional): `yes` to ignore case when matching this expression.

3.9.7. `<put>`

An action which copies a chunk of data from a source reference to a destination reference, optionally transforming it along the way.

Parent	`<expression>` or `<rexrule>`

Children	none
Attributes	• `src` (required): the source reference to copy from. • `dst` (required): the destination reference to copy to. • `xform` (optional): an xform type twin describing a transformation to applied to the copied data.

3.9.8. `<skip>`

An action which skips further processing of an expression's actions based upon some condition involving data in the report or the match. The following conditions are supported:

- `always`: always true.
- `equal`: true if references `a` and `b` have identical values.
- `not equal`: true if references `a` and `b` do not have identical values.
- `defined`: true if reference `a` is resolvable. Used primarily on report variables.
- `undefined`: true if reference `a` is not resolvable. Used primarily on report variables.

Parent	`<expression>`
Children	None
Attributes	• `cond` (required): the condition to evaluate; if this condition is true, the rest of the actions bound to this expression will not be executed. • `a` (required): The first argument to the condition. • `b` (optional): The second argument to the condition.

3.9.9. `<abort>`

An action which terminates further processing of a report, purging it without emitting and validating it, based upon some condition involving data in the report or the match. The supported conditions are identical to those on the `<skip>` element. *This action is not yet implemented*.

Parent	`<expression>`
Children	None

Attributes

- `cond` (required): the condition to evaluate; if this condition is true, the report will be aborted.
  - `a` (required): The first argument to the condition.
  - `b` (optional): The second argument to the condition.

Chapter 4. Relational Database Normalizer (`tabula`)

4.1. Introduction

Tabula is the database normalizer and publisher. It executes SELECT statements against a database, picks data from certain columns of the resulting dataset, packs said data into AirCERT XML documents and ships them off to a collector via XML/SSL.

In the textual world, we do something like the Unix "tail -f" command to see when there is more for us to process. We cannot do this in the database world without e.g. adding a trigger to the database we're normalizing. We must assume for the purposes of this program that write access to the database is simply not allowed; thus, we need a way to do, effectively, what tail -f would do, namely: decide if there is something new in the database for us to normalize.

Adding to the complexity here is the issue of primary keys. The database of our attention should theoretically be organized such that there is, from our perspective, a unique primary key, which may span multiple columns, and perhaps even multiple tables. We need this key because it's the only way we can tell whether or not we've seen this data before. For the sake of argument, let's say that the primary key is a single column, a sequence number. We then must, effectively, detect situations where sequence numbers have wrapped, have been reused, or where the database has been reset from under us. In all such cases, the only option is for the operator of the data collection system that is producing the data to reset our state so that we can start over.

This dance is of more than theoretical interest. If we do not properly detect e.g. sequence number reuse, it is possible for duplicate data to be reported to the collector, which will cause all subsequent analysis on this data to be suspect, at best. Furthermore, we include, in the AdditionalData area of our report, the raw value(s) of the "native" primary key (vis a vis the database we're normalizing); this is so that an analyst can refer back to the original data upon which their analysis is based, e.g. in a telephone conversation with the operator of the data source. We do not necessarily know what the primary key *means* in its native context, however, but we do need to ensure that, when it is needed by humans, it is accurate.

4.2. Installation and Setup

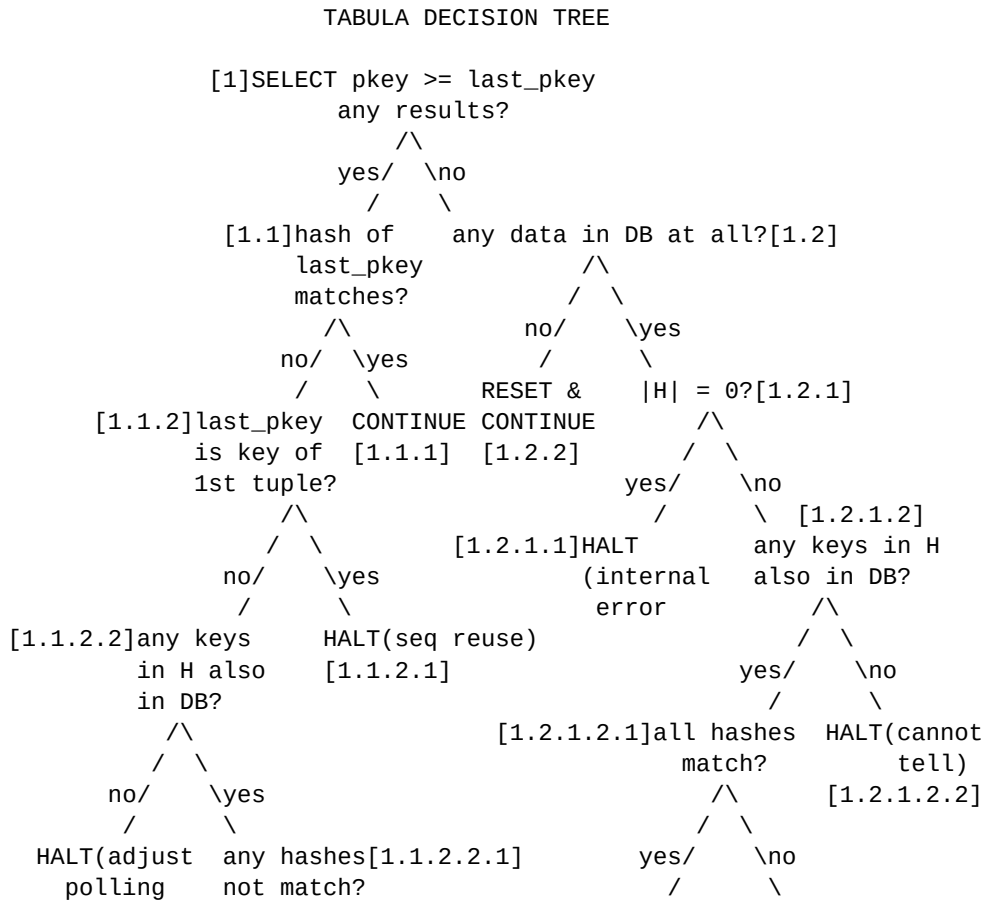
Please see the INSTALL file included in the distribution.

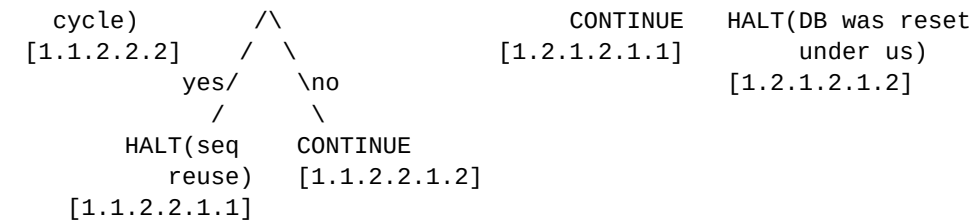
4.3. Running Tabula

Our solution is as follows: All database normalizers keep track of two pieces of information for the databases that they normalize: the primary key from the last row that they processed successfully, and a small list of [primary-key,row-hash[values, where each primary-key in the list is the value of the primary key from a row in the database that we've processed already, and each row-hash is a cryptographic hash (md5) calculated using all the data in the row corresponding to that primary key (including the primary

key itself). We select the rows to put into this list (called H, or the hash list) based on our polling interval; the purpose is to have on hand at least one [primary-key,row-hash] 2tuple that corresponds to a primary key value that is still present in the database (remember, the database we're normalizing is a production data item, it might be periodically purged, backed up, etc.). If there is any doubt as to whether or not the database has been reset behind us, is reusing sequence numbers, etc., the hash list is consulted: we try to find rows in the database that correspond to primary keys in our hash list. For each row that we do find, we compute the hash value from the data that is in the database now, and compare it to the hash value in the hash list. If the values are different for ANY entry in the hash list, then the database is reusing sequence numbers, and an operator must intervene. In practice, we feel that tossing every Nth row into the hash list, where N is something number between 10 and 100, and limiting the size of the hash list to something reasonable (20 perhaps) will be enough for most situations, but in the event that we are normalizing a highly active database that is purged frequently, the defaults may need to be adjusted.

The decision tree for our quasi-"tail -f" algorithm is depicted pictorially below. For clarity's sake, we say that we are selecting data where pkey >= last_pkey, but, really, the primary key may not be a simple number and the query may not be that simple. The idea is that there must be some sort of ordering, and that the values of the primary keys are monotonically increasing through that ordered space, whatever it is. If this is not the case, then the algorithm cannot work. We have not seen such a database schema in practice for the domain that we are addressing (computer/network security), but that doesn't mean that we won't find one somewhere.





The notation HALT(xxx) means that we should communicate xxx back to the collector as an error message, so that the collector's operator can contact the foreign data collection application's operator out of band to correct the issue. CONTINUE means that we are in a normal, monotonically increasing sequence number situation, and should start normalizing from the first row we see from our SELECT statement. RESET means that we should clear our hash list and last_pkey values. If a piece of code below has a comment next to it that mentions a number in square brackets, that means that the code implements the logic behind that part of the decision tree; most of this is in the decision_tree() routine.

The main flow of logic is as follows:

```
configure
connect to collector
for each data source
    connect to data source
    for each ruleset
        for each rule
            result set <- do decision tree
            if decision tree ok
                create new, empty report
                for each row in result set
                    row used <- normalize row into report
                    if report is ready
                        send report
                        create new, empty report
                        if row not used
                            move cursor back one to use for next report
                if report is not empty
                    send report
            if rule shorcircuits and there was no error
                break out of rule loop
    disconnect from data source
disconnect from collector
```

The logic for normalizing a row is:

```
p <- value(s) for primary key column(s) in row
if primary key value in report != p
    if report is empty
        set primary key value in report to p
    else
        return: report is ready, reuse this row for next report
```

```
for each column in the rule that is not primary key
    v <- column's value in the row
    path <- column's path in report
    if v is null
        if column is required
            error: required column is null
    if column does not allow multiple values
        if path already has been set
            error: multiple values for path are illegal
        else
            set path's value in report to v
    else
        add v as value for path
return: report is not ready
```

One could argue that there are many possible ways of making this more efficient, and one would probably be right.

4.4. Configuration

Most (but not all) options can be specified on the command line; they can also be specified in the configuration file. If specified in the config file, they appear as <option> elements inside of the <options> element at top level. For instance:

```
<normalizer>
 <options>
  <option name="working-directory">/tmp</option>
  <!-- ... -->
 </options>
 <!-- ... -->
</normalizer>
```

The following table summarizes the config-file and command-line forms of each option we pay attention to:

```
CONFIG FILE        COMMAND LINE    DESCRIPTION
N/A                --config        config file (./tabula-config.xml)
log-file           --logfile       file to log to, if any
log-syslog         --syslog        bool: log to syslog
log-facility       --facility      log facility to use for syslog
log-stderr         --stderr        bool: log to stderr
log-console        --console       bool: log to console (implies syslog)
log-level          --loglevel      integer verbosity level (0)
ident              --ident         ident for logging
test-mode          --test          run in test mode (no collector)
background         --background    run in background
working-directory  --chdir         dir to chdir to before starting
```

```
xml-schema          N/A             one of: IODEF, IDMEF or SNML (SNML)
collector           --collector     URL to collector
```

Our logging is flexible; we use the libair logging module. We can simultaneously log to a file, stderr, syslog or the system console; we use syslog to log to the console, so console implies syslog, but everything else is independent. This can be handy for debugging, when you want to e.g. turn on logging to stderr to diagnose a problem, and then go back to only using syslog.

The XML schema can only be specified in the config file because it doesn't make sense to mix and match XML schema with config files: generally, a config file will be deeply tied to a particular XML schema, and if an installation uses more than one XML schema, they will need more than one config file for the same task.

Test mode means that we do not attempt to connect to a collector, and just dump the XML documents that we produce onto stdout instead. We also do not save our state across the run in test mode.

The other command-line arguments have to do with the <crypto> configuration:

```
ELEMENT             COMMAND LINE    DESCRIPTION
<private_key>       --privkey       File containing PEM-encoded priv. key
<certificate>       --certificate   File containing X.509 our certificate
```

You cannot specify other certificates (e.g. the collector's) on the command line, they have to be in the config file.

Finally, there must be at least one argument beyond the options, which is a URL specifying the DSN we wish to normalize. These look like:

```
<database>://[<user>[:password]@]<host>[:<port>]/<dbname>
```

e.g.

```
postgresql://sensor:password@localhost:5432/snortData
```

Chapter 5. DAV Transmitter (`dredge`)

5.1. Introduction

Dredge is the AirCERT WebDAV client, the sensor-side communications link to an AirCERT DAV collector, which aggregates incident data for analysis. It provides a simple, reliable method to send a collection of files via DAV/TLS to a specified collector.

5.2. Installation and Setup

5.2.1. Prerequisites

Dredge requires the AirCERT common library (libair), version 0.5.20 or later. libair is available from http://aircert.sourceforge.net/libair, or in the `lib` directory of the AirCERT full source distribution.

5.2.2. Building Dredge

As with all AirCERT applications, Dredge's build system is autoconf-based; therefore, `./configure ; make ; sudo make install` should work to build and install Pathogen. Dredge's build system is tested on Fedora Core (Linux) and Mac OS X.

5.2.3. Obtaining and Installing Certificates

Communications between Dredge and the collector are secured using SSL client certificates. Dredge requires the following two files in order to establish a secure connection with the collector:

- A *CA certificate*, identifying the CA that issued the collector's server certificate and the dredge client certificate. When sending information to the central AirCERT collector at collector1.air.cert.org, this will be the CA certificate belonging to the AirCERT Certificate Authority.
- The *client certificate* belonging to this instance of Dredge, issued by the CA and identified with this instance of Dredge by the collector. This must be a PKCS#12 encoded, unencrypted client certificate containing the certificate as well as the key. When sending information to the central AirCERT collector at collector.air.cert.org, use the openssl pkcs12 command to process the certificate signed by the AirCERT Certificate Authority and the private key identified with this instance of Dredge into a PKCS#12 encoded file.

5.3. Configuring Dredge

Dredge's configuration options may be specified either using an XML configuration file or via the command line.

A Dredge configuration file is an XML file whose top-level element must be `<aircert-config>`, which in turn must contain exactly one `<dredge>` tag. The `<dredge>` tag contains exactly zero or one `<options>` tag followed by exactly one `<crypto>` tag; these delimit the crypto and options sections, respectively. See the figure below for an example Dredge configuration file.

Figure 5-1. Example Dredge configuration file

```
<aircert-config>
  <dredge>
    <options>
      <option name="url" value="https://collector1.air.cert.org/collector"/>
      <option name="in" value="/var/aircert/dredge_in/*.xml"/>
      <option name="fail" value="/var/aircert/dredge_fail"/>
      <option name="retry" value="/var/aircert/dredge_retry"/>
    </options>
    <crypto>
      <certificate encoding="pem" type="x509" use="issuer">
        /usr/local/share/air/aircert-ca.crt
      </certificate>
      <certificate encoding="pem" type="pkcs12" use="client">
        /usr/local/share/air/my-client.p12
      </certificate>
    </crypto>
  </dredge>
</aircert-config>
```

5.3.1. The Options Section

The options section (within the `<options>` tag) consists of one or more `<option>` tags. Each of these sets the option named by its `name` attribute to the given `value`. Each of the available options corresponds to a command-line argument, as shown in the table below.

Table 5-1. Dredge Configuration Options

Option	Command-line	Description
none	`--config`	Name of the configuration file to use. Configuration file options can be overridden on the command line.
`url`	`--url`	URL of the collector. Recommended to be in the configuration file. Required.

Option	Command-line	Description
in	--in	Input file glob expression. Dredge will send all files matching this expression. Recommended to be in the configuration file. Required.
out	--out	Directory to move successfully sent files to. Successfully sent files are left in place by default. Recommended to be in the configuration file.
	--delete-out	Delete successfully sent files.
fail	--fail	Directory to move unsuccessfully sent files to. Unsuccessfully sent files are left in place by default. Recommended to be in the configuration file.
	--delete-fail	Delete unsuccessfully sent files.
retry	--retry	Directory to use as a retry cache for transient send failures. Files are moved into the retry cache when transient failure (route down, etc.) is detected, and transmitted after the failure ends.

Options given in the options section of the configuration file which have corresponding command-line arguments are overridden by the values given on the command line.

5.3.2. The Crypto Section

The <crypto> tag contains one <certificate> tag each for the CA certificate and the client (Dredge instance) certificate. See the example in Figure 5-1 for more detail.

Note that only PEM-encoded CA certificates and PKCS#12 client certificates are presently supported; therefore the encoding attribute on <certificate> elements must be pem for CA certificates and pkcs12 for client certificates. In addition, only X.509 certificates are presently supported; consequently, the type attribute on <certificate> elements must be x509.

All crypto section tags may be overridden on the command-line as follows:

Table 5-2. Dredge crypto options

Option	Crypto section tag	Description
--ca-crt	certificate use=issuer encoding=pem	CA certificate

Option	Crypto section tag	Description
--client-crt	certificate use=client encoding=pkcs12	Dredge client certificate

5.4. Running Dredge

Dredge takes a set of command-line arguments, outlined in Table 5-1 above, as follows:

dredge { --config *config-file* } [--uri *collector*] [--in *in-glob*] [--out *out-directory*] [--delete-out] [--fail *fail-directory*] [--delete-fail] [--retry *retry-cache-directory*] [--lock] [--cleanup] [--daemon] [--pidfile *pidfile*] [--cyctime *input-cycle-time*] [--ca-crt *x509-certificate*] [--client-crt *x509-certificate*] [--loglevel *level*] [--stderr] [--console] [--logfile *log-file*] [--syslog *log-facility*] [--ident *log-identity*] [--help] [--version] [--copyright]

As Dredge runs, it opens each input file matching the specified glob expression and copies it to the collector via WebDAV. Successfully processed and failed files are moved out of the input path after processing. Transient failures are cached in the retry cache in --retry mode. If --lock is specified, lockfiles are created on the WebDAV server before each file is transmitted and removed after the copy is complete; this allows applications on the collector end of the connection to avoid touching files still in transit. In --daemon mode, Dredge continually re-evaluates its input glob expression; otherwise, it terminates after a single processing run. If --cleanup is specified, Dredge removes stale pidfiles and everything from the fail directory at startup time.

By default, Dredge produces no logging output. The following options control how Dredge will log its operation:

Table 5-3. Dredge logging options

Option	Argument	Description
--loglevel	one of panic, critical, error (default), alert, warning, notice, info, debug	log level (less to more verbose)
--stderr	none	log to standard error stream
--console	none	log to system console
--logfile	log file name	log to specified file
--syslog	syslog facility name	log to the specified facility using syslog(3)
--ident	syslog identity (default rex)	set logging identity

Chapter 6. Binary Flow Processor (cargogen)

6.1. Introduction

Cargogen is the AirCERT RDBMS binary flow collector processor, an application which transforms flow data in a supported binary flow format (currently, only Argus RA) into tuples in a supported relational database schema (currently, only AirCERT v1, or ACID 1.x compliant with flow extensions). It is the

6.2. Installation and Setup

6.2.1. Prerequisites

Cargogen requires the AirCERT common library (libair), version 0.5.20 or later. libair is available from http://aircert.sourceforge.net/libair, or in the `lib` directory of the AirCERT full source distribution. As it requires a connection to a database to do its work, Cargogen requires a libair built with at least one RDBMS driver.

6.2.2. Building Cargogen

As with all AirCERT applications, Cargogen's build system is autoconf-based; therefore, `./configure ; make ; sudo make install` should work to build and install Cargogen. Cargogen's build system is tested on Fedora Core (Linux) and Mac OS X.

6.3. Running Cargogen

Cargogen is designed to run either as a one-shot standalone application or as a daemon. It reads each binary flow file from an input glob expression and inserts it into the database. Input file collections are specified as Unix glob expressions. In one-shot mode, after processing all of its input, Cargogen exits; while in daemon mode, Cargogen continually re-evaluates its input glob expression, to support automatic processing of reports dropped into a particular filesystem path (for example, by the AirCERT DAV collector infrastructure).

cargogen { `--in` *`input-glob`* } [`--db` *`database-URL`*] [`--sid` *`sensor-id`*] [`--out` *`output-directory`*] [`--delete-out`] [`--fail` *`failure-out-directory`*] [`--delete-fail`] [`--lock`] [`--cleanup`] [`--daemon`] [`--pidfile` *`pid-file`*] [`--loglevel` *`level`*] [`--stderr`] [`--console`] [`--logfile` *`log-file`*] [`--syslog` *`log-facility`*] [`--ident` *`log-identity`*] [`--help`] [`--version`] [`--copyright`]

Table 6-1. Cargogen options

Option	Argument	Description
`--in`	input file glob	glob expression describing input file paths
`--db`	database URL	URL of database to insert flows into
`--sid`	sensor ID	Sensor to associate inserted flows with
`--out`	output directory	directory to move successfully processed files to
`--delete-out`	none	delete successfully processed files
`--fail`	failure output directory	directory to move processing failures to
`--delete-fail`	none	delete processing failures
`--lock`	none	use .lock files
`--cleanup`	none	remove files in fail destination and overwrite PID file on startup
`--daemon`	none	run in daemon mode
`--pidfile`	PID file file	log process ID to file in daemon mode

As Cargogen runs, it opens each input binary flow file matching the input file glob expression. It then reads each flow record in the file, and translates and writes that record into the database. As each input file is successfully processed, it is moved to the output directory (or deleted, if `--delete-out` is specified; or simply left alone, if neither is specified). Each input file that fails to process is moved to the failure output directory (or deleted, or left alone).

If Cargogen is run in `--daemon` mode, it will continually reevaluate its input glob expression; if not, it will run through each input file matching the glob once, then terminate.

By default, Cargogen produces no logging output. The following options control how Cargogen will log its operation:

Table 6-2. Cargogen logging options

Option	Argument	Description
`--loglevel`	one of panic, critical, error (default), alert, warning, notice, info, debug	log level (less to more verbose)
`--stderr`	none	log to standard error stream
`--console`	none	log to system console
`--logfile`	log file name	log to specified file
`--syslog`	syslog facility name	log to the specified facility using syslog(3)

Option	Argument	Description
`--ident`	syslog identity (default rex)	set logging identity

Chapter 7. Relational Database Denormalizer (`pathogen`)

7.1. Introduction

Pathogen is the AirCERT RDBMS "denormalizer", an application which transforms incident data in one of three XML interchange formats (SNML, IDMEF, IODEF) into tuples in an an arbitrary relational database schema. Pathogen's actions are controlled by an interpreted language unimaginatively named PIL (for "Pathogen Interpreted Language"); Pathogen ships with a collection of PIL programs for handling common interchange format to relational schema mappings.

7.2. Installation and Setup

7.2.1. Prerequisites

Pathogen requires the AirCERT common library (libair), version 0.5.20 or later. libair is available from http://aircert.sourceforge.net/libair, or in the `lib` directory of the AirCERT full source distribution. As it requires a connection to a database to do its work, Pathogen requires a libair built with at least one RDBMS driver.

7.2.2. Building Pathogen

As with all AirCERT applications, Pathogen's build system is autoconf-based; therefore, `./configure ; make ; sudo make install` should work to build and install Pathogen. Pathogen's build system is tested on Fedora Core (Linux) and Mac OS X.

7.2.3. Configuring Pathogen via PIL

Pathogen is essentially an interpreter for PIL. See the "Using the Pathogen Interpreted Language" manual chapter for information on writing PIL programs. Pathogen ships with a collection of PIL programs for dealing with common interchange format to relational schema mappings. These programs are installed into *`prefix`*`/etc` when Pathogen is installed.

Table 7-1. PIL programs supplied with Pathogen

Filename	Input DTD	Output schema
`snml.pil`	SNML 0.3	mod_air 0.9.8/ACID 1.x

7.3. Running Pathogen

Pathogen is designed to run either as a one-shot standalone application or as a daemon. It runs a PIL program over each report in a collection of input reports in an XML interchange format. Input file collections are specified as Unix glob expressions. In one-shot mode, after processing a collection, Pathogen exits; while in daemon mode, Pathogen continually re-evaluates its input glob expression, to support automatic processing of reports dropped into a particular filesystem path (for example, by the AirCERT DAV collector infrastructure).

pathogen { `--program` *`PIL-program-file`* } { `--in` *`input-glob`* } [`--out` *`output-directory`*] [`--delete-out`] [`--fail` *`failure-out-directory`*] [`--delete-fail`] [`--lock`] [`--cleanup`] [`--daemon`] [`--pidfile` *`pid-file`*] [`--loglevel` *`level`*] [`--stderr`] [`--console`] [`--logfile` *`log-file`*] [`--syslog` *`log-facility`*] [`--ident` *`log-identity`*] [`--help`] [`--version`] [`--copyright`] [*`key0=val0`*] [*`key1=val1`*] [*`...`*] [*`keyn=valn`*]

Table 7-2. Pathogen options

Option	Argument	Description
`--program`	program file	PIL program to run
`--in`	input file glob	glob expression describing input file paths
`--out`	output directory	directory to move successfully processed files to
`--delete-out`	none	delete successfully processed files
`--fail`	failure output directory	directory to move processing failures to
`--delete-fail`	none	delete processing failures
`--lock`	none	use .lock files
`--cleanup`	none	remove files in fail destination and overwrite PID file on startup
`--daemon`	none	run in daemon mode
`--pidfile`	PID file file	log process ID to file in daemon mode

In addition to options, Pathogen takes a space-separated list of key=value pairs which are passed to the PIL program as extern strings. The included snml.pil program requires two of these pairs, db= (the libair db URL of the database to connect to) and sid= (the sensor ID of the reporting sensor).

As Pathogen runs, it opens each input report matching the input file glob expression. It then runs the given PIL program over the input file, which causes the contents of the input report to be written into the database for which the PIL program was written. As each input file is successfully processed, it is moved to the output directory (or deleted, if `--delete-out` is specified; or simply left alone, if neither is specified). Each input file that fails to process is moved to the failure output directory (or deleted, or left alone).

If Pathogen is run in `--daemon` mode, it will continually reevaluate its input glob expression; if not, it will run through each input file matching the glob once, then terminate.

By default, Pathogen produces no logging output. The following options control how Pathogen will log its operation:

Table 7-3. Pathogen logging options

Option	Argument	Description
`--loglevel`	one of panic, critical, error (default), alert, warning, notice, info, debug	log level (less to more verbose)
`--stderr`	none	log to standard error stream
`--console`	none	log to system console
`--logfile`	log file name	log to specified file
`--syslog`	syslog facility name	log to the specified facility using syslog(3)
`--ident`	syslog identity (default rex)	set logging identity

7.4. Introduction to PIL

> **Note:** This manual describes a quickly evolving language. There is no expectation that PIL will exist in its present form in the intermediate term. Also given the high variability of the language, this manual section is more "quick and dirty" than the rest of the AirCERT documentation. If you really have a pressing need to write PIL programs for this version of Pathogen, please contact Brian Trammell at bht@cert.org.

Pathogen is essentially an interpreter for a language unimaginatively named PIL (or "Pathogen Interpreted Language"). PIL allows data transformation operations to be expressed to the Pathogen engine in a high-level notation inspired by the syntax of C.

A PIL program is a set of functions and declarations whose main entry point is run once for each input report given to Pathogen. Through manipulation of and iteration over elements of its input report, and execution of SQL statements, it transforms its input reports and inserts them into a database.

Future AirCERT releases will feature a more generalized data transformation engine based upon the Pathogen core, and will be programmed with a language slated to evolve from PIL called SMA (or "Symbolic Manipulation for AirCERT"). This new SMA interpreter will be responsible not only for insertion of reports into a relational database, but also for normalization of information from a variety of formats into the reports in the first place.

7.5. Declarations

A PIL program is made up of a collection of declarations, at least one of which must be an extern tree

declaration (specifying the input XML tree), and at least one of which must be a function declaration named main (the entry point to the program, called once per input tree).

7.5.1. Function Declarations

A function declaration defines a function's entry point, the names of its parameters, and the statements to execute when the function is invoked. A function declaration takes the following form:

Figure 7-1. Function declaration

```
function name (param0, param1, ..., paramn) {
    statement;
    statement;
    statement;
}
```

Statements may only appear in a function body. In order for a PIL program to be valid, at least one function, named main, taking no parameters, must be defined. Parameters are not presently typed; this shortcoming will be addressed in a future revision of the interpreter.

7.5.2. SQL Statement Declarations

A statement declaration defines a parameterized SQL update or query statement to be run against a given database. The statement's parameters are named in the declaration, and their values are substituted for placeholders delimited by $(and) in the statement body at statement invocation time. The URL of the database on which the statement will be run is bound at interpreter start time, and is retrieved from a declared extern string symbol named in the statement. A statement declaration takes the following form:

Figure 7-2. Statement declaration

```
statement name (param0, param1, ..., paramn) on dburlname {
   SQL STATEMENT WITH %(paramn) PARAMETERS
}
```

Warning: Presently, parameters are positional, not named. You must ensure that the order of the parameters in the parameter list matches the order of the parameters in the SQL statement. This is a bug in the runtime and will be addressed in a future release.

7.5.3. Identifier Reverse Map Declarations

A revmap declaration defines a cached map between a set of columns in a database table which uniquely identify a tuple in that table and a unique integer identifier for that tuple. It is most often used to support code tables and referential integrity in one-to-many relationships. A revmap declaration contains a set of

key columns (which uniquely identify a tuple in the table, and map to the integer identifier), the name of an extern string containing a database URL (as with statements), the name of a sequence out of which integer identifiers are assigned, and up to three SQL statements used to maintain the map, labeled preload, select, and insert. The structure of a revmap declaration is as follows:

Figure 7-3. Reverse map declaration

```
revmap name (key0, key1, ..., keyn) on dburlname seqname idsequencename {
    preload { SQL STATEMENT WITHOUT PARAMETERS }
    select { SQL STATEMENT WITH %(keyn) PARAMETERS }
    insert { SQL STATEMENT WITH %(keyn) PARAMETERS }
}
```

The preload statement in a revmap declaration, if present, is run at interpreter initialization time to preload the in-memory cache. For a revmap with n key columns, it must return a result set with n+1 columns. The first column must be the integer ID, and the following columns must be the key columns in the order they appear in the key columns list.

The select statement in a revmap declaration, if present, is run when a key lookup misses to attempt to load an integer identifier for the key column values from the database. If the table behind a revmap is extremely large, this mechanism can be used to avoid preloading.

Ths insert statement in a revmap declaration, if present, is run when a key lookup misses (and a select attempt, if present, misses as well) to add the key column values to the table and return a new integer identifier associated with those values.

7.5.4. Option Declarations

An option declaration sets an interpreter runtime option. It takes the form option name "value";. Currently, only one option is supported, and it is required: the dtd option, which sets the DTD of the input report, and must be one of "snml", "iodef", or "idmef".

Option declarations will be used along with extern declarations in future, more generalized iterations of the system to set input and output context at interpreter start time.

7.5.5. Extern Declarations

Extern declarations define symbols whose values are bound external to the interpreter. They take the form extern type name;. Currently, Pathogen only supports two extern declaration types. There must be exactly one extern tree declaration, which names the variable through which Pathogen will pass its input report as a tree. There may be any number of extern string declarations (for nontrivial PIL programs, there must be at least one to hold the database url name), which name variables that must be set using key=value notation on the pathogen command line.

7.6. Symbols

PIL is a symbolic language; that is, all of its work is done by manipulation of values referenced by symbols. Symbols may be simple (as are strings) or compound (as are trees and result sets). Simple symbols have a single value, which may be accessed or set directly using the name of the symbol, while compound symbols must be accessed via a subscript following the symbol name, enclosed in square brackets. Subscripted compound symbols may be treated as simple symbols for purposes of assignment. This section discusses the types of symbols available in PIL and how to use them.

7.6.1. Literal Strings

Literal strings are simple values, expressed as strings surrounded in double quotes. They cannot be subscripted or assigned to. A double quote character may be included in a literal string by escaping it with a backslash. No other backslash escapes are presently supported.

7.6.2. Variable Strings

Symbols declared as type string hold a single string value. They cannot be subscripted. Strings are the only simple scalar variable type currently available in PIL.

7.6.3. Tree Access

Symbols declared as type tree hold an XML document as a tree. Values in their nodes and attributes may be accessed via air XML paths in the subscript. Subscripted tree symbols are assignable.

7.6.4. Tree Iterator Access

Symbols declared as type treeiterator hold an iterator over the children of a node in a given tree. Tree iterators are created via the builtin new_treeiterator() function. Values in their nodes and attributes may be accessed via relative air XML paths in the subscript. Additionally, each tree iterator has a current node child; use the builtin iterate() function to move the current node child forward. Subscripted access to tree iterators is assignable.

7.6.5. Result Set Access

Symbols declared as type resultset hold an SQL result set. Every result set has a current row; columns in the current row may be accessed via one-based column subscripts. Use the builtin iterate() function to move the current row forward. Subscripted access to result sets is not assignable; result sets are read-only.

7.7. Statements

need front matter about statements here...

7.7.1. Function Invocation

Functions are called using a C-like syntax; the name of the function, followed by the parameters to pass in parentheses. The return value of the function may optionally be assigned to a variable. Function invocation is not presently a true expression; the return value of a function cannot be used directly as a parameter, and it cannot be directly used in a control flow condition, for example. Future versions will fix this shortcoming.

7.7.2. Statement Invocation

Database statements are executed much like functions; the parameters, as with functions, are named in parentheses following the function name. If the called statement is a SELECT statement, it returns a resultset; otherwise, it returns nothing.

7.7.3. Identifier Map Lookup Invocation

Identifier map lookups are invoked much like functions and statements. An identifier map lookup returns the unique integer ID of the row identified by the compound key specified in the parameters as a string. This returned ID is suitable for use as a parameter in a subsequent insert statement, thus allowing Pathogen to construct tables related by integer IDs.

7.7.4. Local Symbol Declarations

Before a symbol can be used in a local function scope, it must be declared and assigned a type. A local symbol declaration consists simply of the name of the type followed by the name of the symbol. Valid types for local declarations are string, tree, resultset, and treeiterator.

7.7.5. Assignment

Values may be directly assigned from one symbol to another. Only assignment of values between simple symbols or subscripted compound symbols is supported.

7.7.6. Conditional execution

PIL supports conditional execution using an if ... elsif ... else construct. The if and elsif clauses take conditions. A condition compares two values for equality or inequality. If a comparison is missing from a condition, the comparison implicitly tests for inequality with the string "0". This is admittedly a little

hackish; a better definition of truth will be supported when simple types beyond strings are available in PIL. One or both sides of a comparison may be an invocation.

A conditional execution construct consists of exactly one if clause, followed by zero or more elsif clauses, followed by zero or one else clauses. Only one of the code blocks in a conditional execution construct will be executed (if no else clause exists, it is possible that none of the code blocks will be executed).

As with much of PIL, the syntax of conditional execution constructs is strongly influenced by C. Conditions are enclosed in parentheses, and code blocks must be enclosed in curly braces, as follows:

Figure 7-4. Conditional execution construct

```
if (condition) {
    statement;
    statement;
} elsif (condition) {
    statement;
    statement;
} else {
    statement;
    statement;
}
```

7.7.7. Iterated execution

PIL provides a while loop construct for iterated execution. While loops use the same condition syntax as conditional execution. The while loop syntax is similarly inspired by C.

7.7.8. Return value assignment

Any simple value may be returned from a function using the return statement. The return statement causes the function to stop processing immediately. While builtin functions (e.g., new_treeiterator) may return compound symbols, user defined functions may not yet do this. This is yet another PIL shortcoming that will be fixed in a future version.

7.8. Built-In Functions

builtin frontmatter

7.8.1. log

The log built-in function takes a single simple value, and writes it to the pathogen application's log. The log function returns nothing.

7.8.2. exists

The exists built-in function takes a subscripted tree or tree iterator and returns the string "1" if the given subscript exists in the tree or current child node, and the string "0" otherwise. It is designed for use from a condition.

7.8.3. xform

The xform built-in function provides an interface to the AirCERT data transformation facility. It takes two parameters, the name of an xform type twin and the simple value to transform. It returns the transformed value, usually for assignment to another symbol. See the Rex manual for more on xform and type twins.

7.8.4. new_treeiterator

The new_treeiterator built-in function creates a tree iterator from a tree or tree iterator. It takes three parameters, a tree or tree iterator to build the new tree iterator on, the path to the parent node whose children to iterate over, and the name of the type of child node to iterate over. This last parameter may be an empty string, which will cause the tree iterator to iterate over all children of the parent node regardless of type. The returned tree iterator's paths are rooted at the current child.

7.8.5. iterate

The iterate built-in function advances the current row in a result set or the current child in a tree iterator. It takes a result set or tree iterator, and returns "1" if there are more children/rows, "0" otherwise. It is intended to be used in a while loop; therefore, it must be called once before accessing the result set or tree iterator.

7.8.6. map_didinsert

The map_didinsert built-in function takes a revmap and returns "1" if the last lookup operation caused a new record to be inserted in the backend, "0" otherwise. It is intended to be used in an if condition to cause ancillary information to be inserted into a database associated with a revmap entry.

7.9. Other syntax tidbits

Comments are C-style; they start with /*, end with */, and may span lines. C++-style comments are not supported.

7.10. Error handling

PIL supports no error-handling primitives in-language. If a processing error occurs, the input file is routed to the fail destination (see the Pathogen manual) and processing continues with the next file.

Chapter 8. The AirCERT Sensor Infrastructure

8.1. Introduction

The AirCERT sensor infrastructure handles the first four stages of the AirCERT workflow - capture, generation, normalization (where appropriate), and transmission. Each sensor is built with a combination of AirCERT components (e.g., rex, dredge, and twrap), open source event generation components (argus and snort), and AirCERT-provided glue. This chapter discusses that glue.

The AirCERT sensor management infrastructure consists of the *AirCERT::SensorMgr* Perl module and associated scripts, *pcap_mgr*, *argus_mgr*, and *snort_mgr*. This module depends on the *AirCERT::GenUtil* Perl module, which provides the *twrap* tarball creator as well as a large collection of general Perl utility subroutines. These modules are built and distributed as CPAN-friendly modules; run `perl Makefile.PL; make; make test; sudo make install` to install them.

8.2. General Principles

Several general principles apply to all the sensor infrastructure scripts. First, each script shares an *air-home* directory; this is generally `/var/aircert` on production AirCERT sensors. Each generator is treated as a different logical sensor; generation is paired with normalization and transmission, and this logical sensor stack runs in its own directory named after the *sensor type* under the air-home directory; for example, the Argus sensor runs under `/var/aircert/argus-rag`. The capture stage is treated as if it had the virtual sensor type *pcap*.

Each logical sensor also has a *next-stage*, that is, the logical sensor that sensor will pass its raw pcap data on to after it is done processing. Each logical sensor also generally has a *cycle* time; the interval between polling directories for available data. This acts as a clock which steps data forward through the sensor stages, and is set at 300 (five minutes) in production AirCERT sensors.

8.3. Capture

Capture is managed by the *pcap_mgr* script, which takes the following arguments:

pcap_mgr [`--tcpdump` *`tcpdump-executable [tcpdump]`*] { `--interface` *`pcap-if-name`* } [`--size` *`pcap-max-size-in-MB [32]`*] [`--cycle` *`cycle-time [300]`*] { `--next-stage` *`sensor-type`* } [`--user` *`next-stage-owner [root]`*] [`--air-home` *`air-home-directory [/var/aircert]`*] [start | stop]

The pcap_mgr script manages the execution of tcpdump, and rotates output files out from tcpdump on completion to the next-stage logical sensor. The `--size` option must be set to ensure that at least one pcap file is written every `--cycle` seconds, or collection rate artifacts may be visible in flow data.[1]

Since tcpdump must often be run as root, the `--user` option allows pcap_mgr to change ownership on output files rotated into downstream logical sensors.

8.4. Flow Event Generation and Normalization

AirCERT uses Argus for flow event generation. The *argus-rag* (for Argus RA binary output, aGgregated) sensor type is managed by *argus_mgr*, which takes the following arguments:

argus_mgr [`--ragator-cycle` *`aggregation-window [3600]`*][`--cycle` *`cycle-time [60]`*] [`--next-stage` *`sensor-type [null]`*] [`--user` *`processor-user`*] [`--argus-root` *`argus-prefix-directory [/usr/local]`*] [`--air-bin` *`air-executable-directory [/usr/local/bin]`*][`--air-home` *`air-home-directory [/var/aircert]`*] [start | stop]

While the pcap_mgr `--user` option changes the owner of the output, the argus_mgr `--user` option changes the owner of the generator and transmitter processes. This is useful because the _mgr scripts are designed to be run like SysV init.d startup scripts, and may be run as root, while it is probably a bad idea to run the processors with unnecessary root privileges.

The two cycle times for argus_mgr are for the standard argus pcap data poll and the ragator aggregation period. The aggregation period is subject to a tradeoff - too short an aggregation period and the data may contain a significant count of non-aggregated flows, while too long an aggregation period injects significant delay into reporting. The default is one hour, but production AirCERT sensors presently use five-minute aggregation bins.

Since argus-rag sensors transmit raw Argus binary flow data, there is no normalization stage managed by argus_mgr.

8.5. NIDS Alert Generation and Normalization

AirCERT uses Snort for NIDS Alert event generation. The *snort-snml* (for Snort via SNML) sensor type is managed by *snort_mgr*, which takes the following arguments:

argus_mgr [`--cycle` *`cycle-time [60]`*] [`--next-stage` *`sensor-type [null]`*] [`--user` *`processor-user`*] [`--snort` *`snort-executable-path [snort]`*] [`--air-bin` *`air-executable-directory [/usr/local/bin]`*] [`--air-etc` *`air-config-directory [/usr/local/etc]`*][`--air-home` *`air-home-directory [/var/aircert]`*] [start | stop]

The snort_mgr `--user` option causes a privilege drop, similar to argus_mgr as above.

8.6. Transmission

Each manager script also ensures that twrap and dredge start for each virtual sensor. The important thing to note about transmission for operational purposes is that each logical sensor (sensor machine/sensor type pair) is treated as a separate identity by the collector; that is, that each sensor type's dredge process needs its own client certificate. See the configuration section below for more.

8.7. Configuration

Configuration of a new AirCERT sensor using the AirCERT sensor infrastructure scripts is a matter of installing software, creating a sensor user, creating an Air home directory structure and sensor-local configurations, and installing identities.

The following software is required on the sensor side:

- tcpdump
- argus 2.0.6
- argus-clients 2.0.6
- snort 2.2.0 ot later
- Perl 5.8 or later
- OpenSSL 0.9.7 or later
- expat 1.95 or later
- neon 0.24.7 or later
- librrd.a 1.0.49 or later
- PostgreSQL 8.0.1 or later (client libraries only)
- libair 0.5.20 or later
- dredge 0.5.10 or later
- rex 0.3.13 or later
- AirCERT::GenUtil 0.76 or later
- AirCERT::SensorMgt (this package) 0.75 or later

The Air home directory structure and sensor-local configuration information are largely created by the *argus_reconf* and *snort_reconf* scripts. These scripts are invoked as follows:

argus_reconf { `--collector` *`url`* } [`--bin-length` *`aggregation-window [3600]`*] [`--air-home` *`air-home-directory [/var/aircert]`*]

snort_reconf { `--collector` *`url`* } [`--air-home` *`air-home-directory [/var/aircert]`*]

Additionally, the `pcap` and `pcap/run` directories must exist in the Air home directory. After these directories have been created, only certificates for Dredge must be installed.

The Dredge PKCS#12-encoded client certificate for a given virtual sensor goes in `${air-home}/${sensor-type}/ssl/clicert.p12`, and the PEM-encoded CA certificate used by the collector and sensors goes in the `cacert.pem` file in the same directory. Maintenance of the public key infrastructure required for a sensor/collector network is beyond the scope of this document.

Notes

1. Since the minimum size of a rotated pcap file is one megabyte, this implies that for the standard five-minute cycle time, the minimum five-minute data rate is 27.3 kilobits per second. This may be a problem for backscatter or honeynet sensors; a solution is forthcoming in a future release.

Chapter 9. The AirCERT DAV Collection Infrastructure

9.1. Introduction

The AirCERT collector infrastructure handles the final stage of the AirCERT workflow, helpfully named Collection. Each collector is built with a combination of AirCERT components (e.g., cargogen, pathogen, and untwrap), open source collection technologies (namely Apache httpd, mod_ssl, mod_rewrite, and mod_dav), and AirCERT-provided glue. As with the preceding sensor management chapter, this chapter discusses that glue.

The AirCERT collector management infrastructure consists of the the *AirCERT::DavMgr* Perl module and associated scripts, *dav_mgr*, *dav_reconf*, *dav_sensoradd*, and *dav_sensordel*. This module depends on the *AirCERT::GenUtil* Perl module, which provides the *untwrap* tarball extractor as well as a large collection of general Perl utility subroutines; and the *AirCERT::CollectorDB* module which provides access to the sensor information tables in an AirCERT database. These modules are built and distributed as CPAN-friendly modules; run `perl Makefile.PL; make; make test; sudo make install` to install them.

9.2. General Principles

The general principles applicable to collector management are similar to those on the sensor side. One difference is that the *air-home* working directory must contain a `collector` directory to contain collector working files. Depending on your WebDAV configuration, it may also contain a `dav` directory to contain the inbound DAV files.

9.3. Installation and Configuration

The AirCERT collector uses Apache httpd to receive transmitted tarballs from the sensors via WebDAV over TLS. The mod_rewrite module is additionally used to map the same URL to different paths within the air-home dav directory based upon the client certificate. This allows each sensor to have read-write access to their own DAV receiving directories without exposing other sensors to the dangers of that read-write access. The dav and mod_rewrite configuration and directory heirarchies are managed by the dav_reconf script.

First, ensure that all the required collector software is installed:

- Perl 5.8 or later
- OpenSSL 0.9.7 or later
- expat 1.95 or later
- neon 0.24.7 or later

- librrd.a 1.0.49 or later
- PostgreSQL 8.0.1 or later (client libraries only)
- Apache 1.3.x or 2.x with mod_ssl, mod_dav, and mod_rewrite
- libair 0.5.20 or later
- pathogen 0.5.10 or later
- cargogen 0.2.6 or later
- DBI.pm and DBD::Pg.pm
- Crypt::OpenSSL::X509
- AirCERT::GenUtil 0.76 or later
- AirCERT::CollectorDB 0.70 or later
- AirCERT::DavMgt (this package) 0.76 or later

Before configuring Apache, you'll need to create PEM-encoded certificates for each virtual sensor your collector will receive data from, then use the dav_sensoradd script to add those sensors to the collector database, as follows:

dav_sensoradd { `--db` *`database-url`* } { `--certificate` *`client-pem-cert`* } { `--host` *`sensor-hostname`* } { `--interface` *`sensor-pcap-interface`* } [`--filter` *`sensor-pcap-filter [null]`*] { `--type` *`sensor-type`* } { `--prefix` *`sensor-prefix`* }

The `--prefix` option is a way to assign a short name to a machine containing one or more virtual sensors; it appears only in local dav and collector paths, and is provided as an ease-of-administration tool.

To remove a sensor, use dav_sensordel:

dav_sensordel { `--db` *`database-url`* } { `--certificate` *`client-pem-cert`* }

Once all the sensors have been added to the database, use dav_reconf to reconfigure apache and the collector paths:

dav_reconf { `--db` *`database-url`* } [`--air-home` *`air-home-directory [/var/aircert]`*] [`--dav-home` *`dav-home-directory [/var/www/dav]`*] [`--apache-home` *`apache-serverroot-directory [/etc/httpd]`*] [`--rrd-mode` *`processor-rrd-output-mode [off]`*]

This will create the appropriate directories as necessary under air-home and dav-home, and will additionally create an apache configuration file in conf/air/airdav.conf inside the Apache ServerRoot directory. This file must be included within the SSL virtual host directive in order to enable DAV and rewrite as appropriate. Additionally, the main Apache configuration file must enable the required modules (as above).

9.4. Collection

Collection is composed to two steps: receipt and processing. Data receipt is handled by Apache; to start receiving and spooling data, simply start Apache. Processing is managed by the dav_mgr script, which is run as follows:

dav_reconf { `--db` *`database-url`* } [`--user` *`processor-user`*] [`--air-home` *`air-home-directory [/var/aircert]`*] [`--dav-home` *`dav-home-directory [/var/www/dav]`*] [`--air-bin` *`air-executable-directory [/usr/local/bin]`*] [`--air-etc` *`air-configuration-directory [/usr/local/etc]`*] [`--rrd-mode` *`processor-rrd-output-mode [off]`*] {start | stop}

`--air-home` and `--dav-home` must match those used in dav_reconf. `--rrd-mode` may be `basic` or `ext`, for causing processors to produce inline RRD output from processing for monitoring purposes, and should also match that passed to dav_reconf.

`--user`, as with sensor_mgr, is used to cause the dav_mgr script to drop permissions on startup. The dav directory must be writable both by the Apache server user and the processor user, and the collector directory in air-home must be writable by the processor user.

Chapter 10. The Cheap AirCERT Visualization Environment (CAVE)

10.1. Introduction

The AirCERT release includes a simple periodic analysis frontend called CAVE (or Cheap AirCERT Visualization Environment). CAVE is designed to pipe data from arbitrary SQL queries into arbitrarily specified RRDtool round-robin database files, to pipe data from arbitrary RRDs through an external process to other RRDs, and to draw arbitrary graphs from the resulting RRDs. It ships with a configuration file that provides a set of simple analyses.

CAVE is a CPAN-friendly Perl module (AirCERT::CAVE); run `perl Makefile.PL; make; make test; sudo make install` to install it. After installing CAVE, run `perldoc AirCERT::CAVE` to read the authoritative CAVE documentation.

10.2. Running CAVE

CAVE, like the infrastructure scripts, is built around the concept of a working directory. A CAVE working directory contains a configuration file, *caveconfig.pl*, and one directory per *task* defined in that file. CAVE ships with two entry points. *caverun* runs a given task in a given CAVE working directory, and *cavereconf* generates derivative configuration information from the caveconfig.pl file in a given CAVE working directory (currently, this is only the *cavejax.xml* file used by the CAVE/AJAX frontend).

The output of a task run with caverun is a collection of PNG files in the given task directory in the CAVE working directory. Each task should be run periodically from cron, optionally followed by a command to move the output to a web server from which cavejax.xml and cavejax.html (the CAVE/AJAX frontend) are also available.

The supplied configuration file contains a set of simple analyses. Information on editing that configuration file is available from the perl POD documentation supplied with the module.

Chapter 11. The AirCERT Common Library (`libair`)

11.1. Introduction

Libair is a set of common utilities used by the rest of the AirCERT project. The functionality is broken down into four main modules:

- xml -- DTD-driven XML parsing and generation code
- db -- database access abstraction code
- xform -- generic data transformation engine
- util -- utility functions (e.g., logging, memory allocation)

At this time, libair is meant to be used from C (and possibly C++) programs. There are no run-time binding for Java, perl, Python, or any other language.

The current code-base was designed and tested on Linux, Free/OpenBSD, and Mac OS X, but should be usable (although untested) without change to other unix platforms as well as Windows.

11.2. Installation

Please see the INSTALL file included in the distribution.

11.3. XML Report Handling

(See xml/README for additional details)

The xml component is the heart of libair. It is a DTD-driven XML engine that uses a template approach to building up a data structure intended to conform to one of the supported DTDs. The two main data structures exported by this component air air_xml_tree_t and air_xml_node_t.

For tree construction, our approach is to automatically generate the C code necessary to glue together a data structure based on an arbitrary DTD. When a client application creates an air_xml_tree_t, it must specify which DTD the tree conforms to; the resulting (opaque) data structure is a fully-formed instance of this DTD, in as much as it is possible to instantiate it from scratch. Thereafter, all modifications to the tree are done via paths ala XPATH; depending on the DTD, setting a path in a tree may cause intermediate nodes to be instantiated on the fly, and default values be given to their attributes. At any point during the construction of a tree, the validator may be invoked on it to see if it is a legal instance of the DTD. Any well-formed tree can be turned into a string representation that is legal XML, and which will parse using the libair xml's parsing operations (which use the lower-level tree construction and modification routines to do their work). Schema support is planned, as is run-time addition of new DTDs and Schemas.

Currently, four DTDs are supplied with libair; the support for these DTDs is built in at compile-time, and the specific set of DTDs can be customized via a ./configure option. The supported DTDs are:

- Intrusion Detection Message Exchange Format (IDMEF)
- Incident Object Description Exchange Format (IODEF)
- Simple Network Markup Language (SNMLv3)
- AirCERT-Config

IDMEF is an Internet standard for exchanging security incident meta-data. IODEF is a draft standard that encapsulates IDMEF, and which provides a richer data model for capturing less structured information along with the raw incident data. SNML is a non-standard, but extremely useful DTD that is tied quite closely to the the AirCERT system; it is intended to capture Snort IDS data in a succinct and unambiguous way. Finally, AirCERT-Config is an AirCERT-internal DTD that defines the configuration file language accepted by all of the AirCERT tools.

11.4. Data Transformation

Xform is a type-based data transformation engine used by several AirCERT tools, such as rex, the regular expression normalizer (not a part of libair).

11.5. Database Abstraction Layer

The db component implements a quasi-DBI-like interface to dealing with RDBMSes that hides all of the low-level details. Currently, this component has drivers for MySQL, PostgreSQL, ODBC, and OCI. It also has AirCERT-specific code that provides a higher-level API for AirCERT apps to use, which knows about the AirCERT schema and makes certain commonly used operations simple to use.

11.6. Utility Functions

The util component has a number of sub-modules that provide a number useful, basic primitives.

- ualloc -- user allocation domains (level of indirection for malloc)
- panic -- code to bug out when you need to
- log -- flexible, multi-output logging and debug messages
- cla -- command-line arg parser and usage message generator
- flagbits -- flagword -> string for readable log messages
- url -- URL parsing and construction
- daemon -- utilities for writing daemons
- strutil -- string-bashing code that doesn’t belong anywhere else

- timing -- macros to generate timing information of code execution
- https -- routines to manage https communication
- file -- portable/useful file primitives
- xbuf -- extendable buffers/vectors
- hash -- simple in-core hash tables

Glossary

C

Collector

Some reasonable definition here.

E

Extensible Markup Language (XML)

Some reasonable definition here.

I

Intrusion Detection Message Exchange Format (IDMEF)

Some reasonable definition here.

Incident Report

Some reasonable definition here.

Incident Object Description and Exchange Format (IODEF)

Some reasonable definition here.

N

Network Intrusion Detection System (NIDS)

Some reasonable definition here.

Normalizer

Some reasonable definition here.

S

Security Event

Some reasonable definition here.

Simple Note Markup Language (SNML)

Some reasonable definition here.