“monk-it”

Prism++ Documentation

University of Erlangen
Computer Networks and Communication Systems
and Regional Computing Center

Authors

Tobias Limmer
Falko Dressler
Martin Gründl

23.12.2008

Contents

1. Introduction

The goal of this project is not only to develop a correlation engine, but to create a framework for correlator modules and a simple runtime environment which executes, controls and monitors the components. The developed tool Prism++ is based on earlier work on Prism [KVHD06a, KVHD06b].

The basic architecture of the correlation engine consists of three major layers outlined in Figure 1.1. A receiver collects all incoming events and preprocesses them for further use. These events are then analyzed, filtered and processed according to the various implemented algorithms. Finally the resulting alerts are stored for later reference or more specific analysis and an operator or other systems is notified, if required.

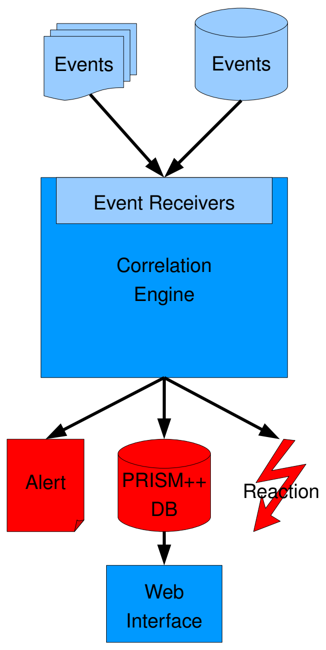


Figure 1.1.: PRISM++ correlation engine overview

While most correlation engines currently available are suited for either analyzing real-time event streams or processing archived events in batch mode, we provide a tool which supports real-time analysis and is able to analyze archived events by simulating real-time operation as well in fast-forward mode. The advantage of this approach is the ability to reconstruct the previously obtained results and to analyze them with regard to a specific aspect while using either the same configuration or a variation of it.

2. Prism++

2.1. Correlation engine

To increase the flexibility of our implementation and to ease the implementation of additional functionality, we realized all components of the correlation engine as independent modules, which are usable at any position in the architecture.

Receiving Events The main task of this layer is to receive events, normalize them according to a common format and optionally archive them for later reference or in-depth analysis. Finally, they are forwarded to the correlation components for further processing.

Correlation To enhance the flexibility of the correlation engine, the actual processing units are based on a common design to allow flexible reconfiguration for additional analysis or future requirements. Consequently, this feature requires standardized internal data structures, which represent the events internally and enable an unrestricted composition of the processing units.

All events coming from the input layer traverse a configured chain of components and each of them performs operations on the event. Besides simple filtering to remove irrelevant events, analyzing specific properties to detect patterns or collecting statistical data, a processing unit may also aggregate multiple related events into a *meta event* to decrease the amount of events. The events may also be enriched with metadata that supplies additional information about involved networks, hosts or services.

Results The results of the correlation process are finally processed by output components, which are responsible for archiving the resulting alerts and all activity pertaining to notification and reaction.

It is especially important to store all alerts generated by the correlation engine permanently for later analysis and to facilitate the reconstruction of incidents or anomalous behavior. If the correlation engine is monitoring a specific, well-known network for intrusions, notification of security personnel about potential intrusions is an important feature as well. Depending on the environment, it may also be suitable to automatically initiate active countermeasures against detected incidents, for example by reconfiguring firewalls to prevent further damage.

To access the alerts reported by the correlation engine and to analyze available data, a web interface is provided which provides access to the alerts generated by the correlation engine and to the events received from sensors as well.

2.1.1. Requirements

The following requirements were used as a foundation for the design and implementation of the PRISM++ correlation engine.

1. **Integration of multiple data sources** A fundamental mechanism of event correlation is the analysis of events coming from multiple data sources and, therefore, it is quite important to facilitate the integration of additional systems into the existing architecture.

2. **Extensibility** As event correlation is currently an active field of research for academic institutions and commercial organizations due to the increasing awareness for security, new approaches for correlating events are constantly developed. To follow this trend, our correlation engine should not only be able to provide meaningful results to analysts, but also actively encourage the security community and other audiences to experiment with the numerous possibilities of event correlation. Therefore, both the architecture and the language used for the implementation should provide a high level of abstraction and hide the internal operations not directly related to the processing of events.

3. **Flexibility** In addition to the development of new modules, which is encouraged by offering simple interfaces, the usage of existing components in new combinations or for different purposes may provide interesting results and new ways to analyze and correlate events. For this reason, it should be easy for both operators and developers to control the configuration of the modules and their interconnections in the correlation engine.

4. **Transparency and Traceability** Because of the complex algorithms used for event correlation and the large amount of data processed by them, it is important to provide an insight into the data flows and internal data structures of the correlation engine. Furthermore, such possibilities also facilitate and thereby encourage the development and debugging of new correlator modules.

5. **Reproducibility** After detecting an anomaly or any suspicious behavior, it is often very useful to analyze the received events with different tools, algorithms or from a different point of view. To support this form of batch analysis on previously received data, the incoming events must be archived for later access. The correlation engine, in turn, should be able to simulate the real-time arrival of these events and process them accordingly.

6. **Integration of metadata** Often the received events do not indicate their importance or relevance for the monitored network in general and the target of the attack in particular. But by using external data sources such as inventory databases or vulnerability scanners, contextual information about an event and the potentially affected devices may be obtained, which improves the assessment of an event's risk or impact.

7. **Aggregation of events** An integral part of any correlation process is the aggregation of events to decrease the amount of data presented to security analysts or operators. Therefore, a correlation engine should provide mechanisms and special data structures for the aggregation of events.

8. **Prioritization of events** Although the previous goal decreases the amount of events presented to an operator, the amount of data may still be too high for manual review. To avoid the unnecessary analysis of events, another very important component of event correlation is the assignment of priorities. Based on these priorities, the operator and optionally any notification or reaction modules are able to decide, whether the respective event requires special attention or even reaction.

2.1.2. Event Representation

After stating the requirements for the PRISM++ correlation engine, we now present its internal representation of events.

To support the utilization of multiple types of sensors, a subset of common attributes must be defined, which abstracts from the different event representations. Otherwise, each correlator module would have to deal with the peculiarities of all supported sensors and their event formats. The definition of such a subset also facilitates the process of implementing support for a new type of data source.

Besides some fundamental attributes such as a timestamp, source and destination addresses, identifiers for the event and a description, support for prioritization and archiving events must be integrated as well.

Whereas the attributes explicitly allowed for each type of event are specified in the documentation, the use of additional attributes is not restricted to facilitate the integration of external data sources such as inventory databases or other metadata.

To represent the aggregation of multiple events based on a similarity metric or other mechanisms, a special type of event, the *meta event*, has been implemented. In addition to the common functionality provided by an event representation, it provides an event container, which is able to store multiple events of arbitrary types. As meta events usually contain multiple events, the meta event representation must be able to handle multiple values for each of its attributes as well.

To ensure maximum flexibility, we also support meta events consisting of meta events and, thus, even a tree-like, recursive structure of events, as depicted in Figure 2.1, may be created.

Table 2.1 shows all supported parameters for events. Additionally, metadata may be stored inside these events as well as a set of references to other events inside meta events.

2.1.3. Architecture of the Correlation Engine

The correlation engine itself consists of instances of correlator modules and interconnections transporting the internal event representations. Of course, some common subsystems are provided as well.

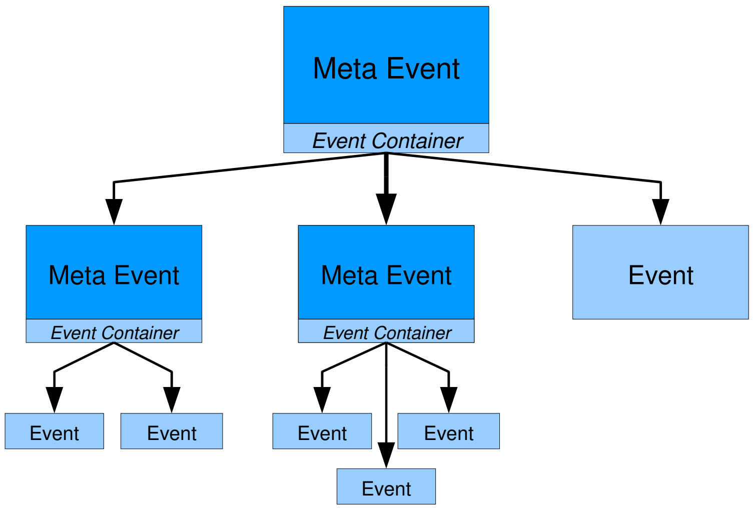


Figure 2.1.: Tree of meta events

The interconnections themselves are realized as simple FIFO[1] queues, which ensure that the correct order of the events is preserved. As these interconnections are freely configurable, creating new configurations and integrating additional modules is quite easy and encourages researchers and security analysts to evaluate various combinations of correlator modules.

Each correlator module has exactly one input queue, which receives the forwarded events from all its predecessors. On the other hand, it must be able to support multiple output queues, for example to distribute events received by data sources to multiple correlator modules in parallel.

On startup, after initializing some common subsystems, the modules are initialized and interconnected according to the specified configuration. As soon as this process has completed, events received by modules representing data sources are forwarded to their respective successors and thus the event flow through the correlation engine is established.

The event flow through the modules of the correlation engine may also be used for distributing signals to all modules, such as a request for termination if the engine shuts itself down or is forced to do so.

The resulting architecture of the complete correlation engine is depicted in Figure 2.2.

Correlator Modules

The decision to encapsulate most functionality of the correlation engine and especially the processing of events in modules was primarily driven by the requirements for flexibility and extensibility.

[1]first in, first out

Attribute name	**Description**
timestamp	Time the event occurred
src_ip	Source IP*(integer)*
dst_ip	Destination IP*(integer)*
ip_protocol	IP protocol number
src_port	Source port for TCP/UDP
dst_port	Destination port for TCP/UDP
icmp_type	ICMP type
icmp_code	ICMP code
prism_event_type	Event type: SNORT — IDMEF — PRISM
prism_table_name	Table the event is stored in
prism_event_id	ID of the event in the table
prism_sig_id	PRISM++'s internal signature ID
sensor_gen_id	Sensor-specific generator ID, e.g. for sensor plugins
sensor_sig_id	Sensor-specific signature ID
sig_name	Signature name
description	Additional information
priority	Priority assigned to the event*(integer)*

Table 2.1.: Attributes of the PrismEventInterface class

While providing a simple, common interface for all correlator modules enables the free arrangement of modules and their interconnections and, therefore, increases the flexibility of this approach, the resulting complexity should be hidden from the user. Consequently, we provided an abstraction layer in the form of a base class for correlator modules, which hides any activities pertaining to the internal operation of the modules, for example the transport of events.

Any activity pertaining to receiving and forwarding event objects are implicitly handled by the implementation of this base class and a single method acts as an interface to all internal activities.

As depicted in Figure 2.3, it gets called for each received event and controls the forwarding of the event by its return value.

In this method, the correlator module may change almost any aspect of the received event. It may add, update or remove attributes to it, run algorithms based on its attributes, influence its priority, cache it for some time to compare it to other events, write the event to persistent storage or obtain additional data from external sources like metadata.

To avoid most issues pertaining the temporal order of events, which is especially relevant for modules with internal caches or time-based correlation, all modules use an internal clock, which is implicitly updated by the base class and provides a consistent time reference for each module.

Data sources for the correlation engine are implemented as a special form of correlator modules as well. While usual correlator modules process and optionally forward the

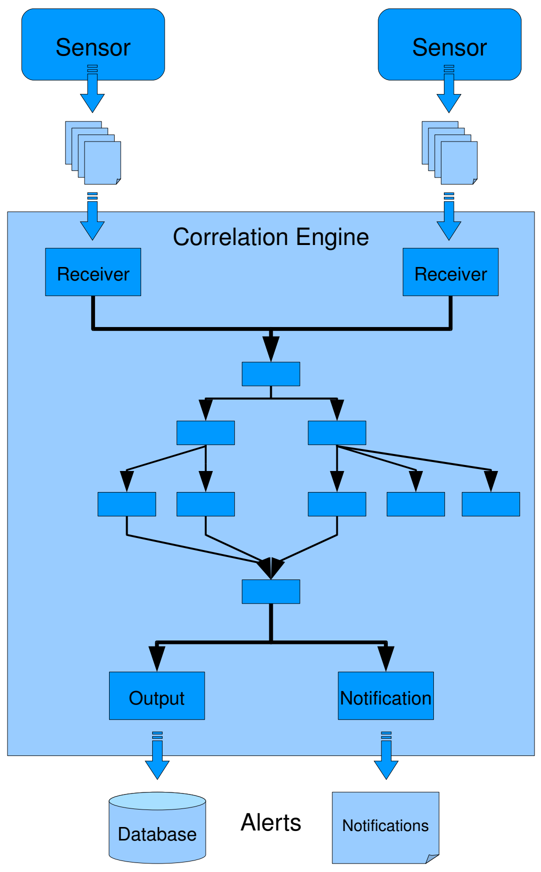


Figure 2.2.: Design of the PRISM++ correlation engine

events they received from other modules, data sources forward events received from the respective sensor and usually do not receive any events from other modules.

Persistence

To store data produced by the correlation persistently, a relational database management system, specifically PostgreSQL, was used. In addition to the alerts generated by the correlation engine, it is also used for archiving the events received from its sensors.

Of course, any correlator module will be able to use the database for its internal data structures or for accessing external data sources as well. Thus we get the benefits inherently provided by a database, such as the powerful query capabilities of SQL[2] as well as efficient storage of data and fast access to large amounts of data by using the database's index functionality.

[2] *Structured Query Language*, a standardized language for retrieving and manipulating database records

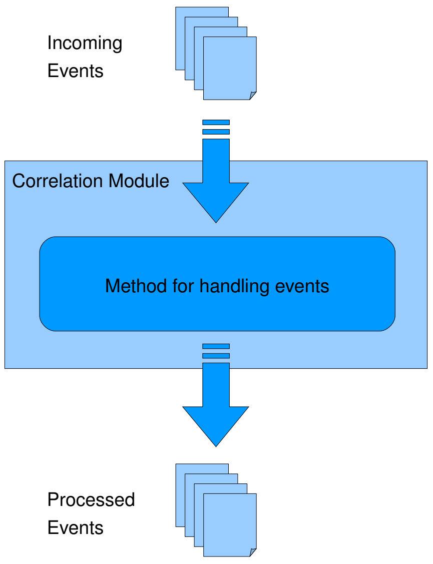


Figure 2.3.: Functionality of a correlator module

2.2. Webinterface

We created a webinterface for managing the Prism++ correlation engine as well as easy manual analysis of incoming events and correlated events. It greatly eases the configuration of the correlation engine, as the configuration of correlation including meta events is an iterative, manual process where the analyst determines best prioritization conditions for the meta events.

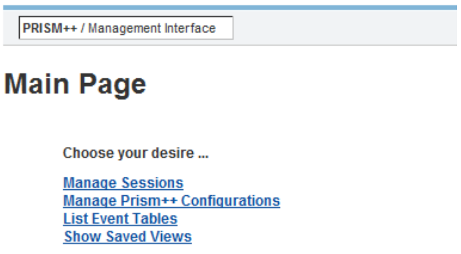
The start screen of the webinterface is displayed in figure 2.4. It is differentiated in two basic parts. One the one hand, the webinterfaces supports management of the correlation engine remotely by editing configurations, and starting and stopping the correlation process. These functions can be accessed by clicking on the links ‘Manage Prism++ Configurations’ and ‘Manage Sessions’.

The second category of functions provides methods for displaying and analyzing events. There, events can be filtered, sorted and grouped to meta events by the analyst. Furthermore, filters and display settings can be saved as ‘views’. On the start screen, these functions can be accessed by the links ‘List Event Tables’ and ‘Show Saved Views’.

2.2.1. Correlation engine management

A session is configured by the user and represents the interface to the correlation engine. The engine’s configuration is defined using a XML file that contains all relevant parameters for used modules, and also information about database access.

The single correlation modules listed in the configuration file have predefined functions, are freely combinable and are executed by the correation engine using multiple threads. The events are written to a database and then they may be read and displayed by the



PRISM++ / Management Interface

Main Page

Choose your desire ...

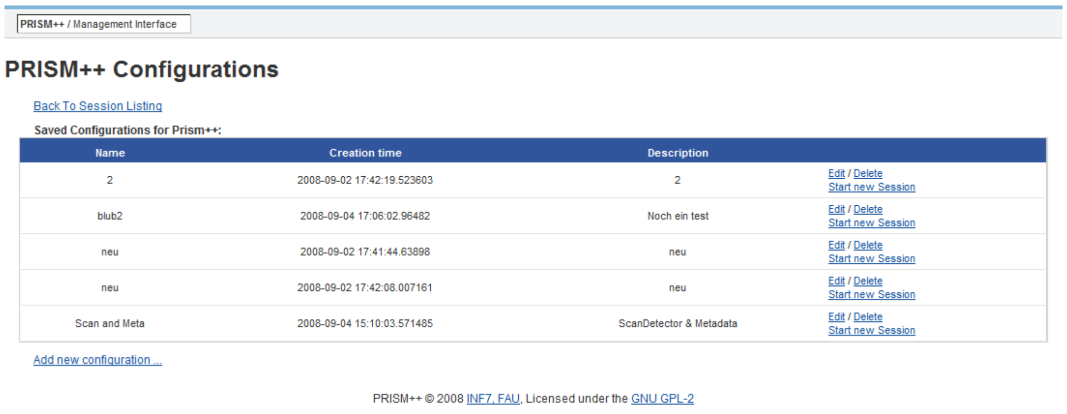
Manage Sessions
Manage Prism++ Configurations
List Event Tables
Show Saved Views

Figure 2.4.: The start screen of Prism++'s web interface

Prism++ web interface.

PRISM++ configurations

Figure 2.5 shows an example of the configuration listing screen within the web interface. Links 'Add new configuration' and 'Edit' allow the user to add and edit Prism++ configurations.



PRISM++ / Management Interface

PRISM++ Configurations

Back To Session Listing

Saved Configurations for Prism++:

Name	Creation time	Description	
2	2008-09-02 17:42:19.523603	2	Edit / Delete Start new Session
blub2	2008-09-04 17:06:02.96482	Noch ein test	Edit / Delete Start new Session
neu	2008-09-02 17:41:44.63898	neu	Edit / Delete Start new Session
neu	2008-09-02 17:42:08.007161	neu	Edit / Delete Start new Session
Scan and Meta	2008-09-04 15:10:03.571485	ScanDetector & Metadata	Edit / Delete Start new Session

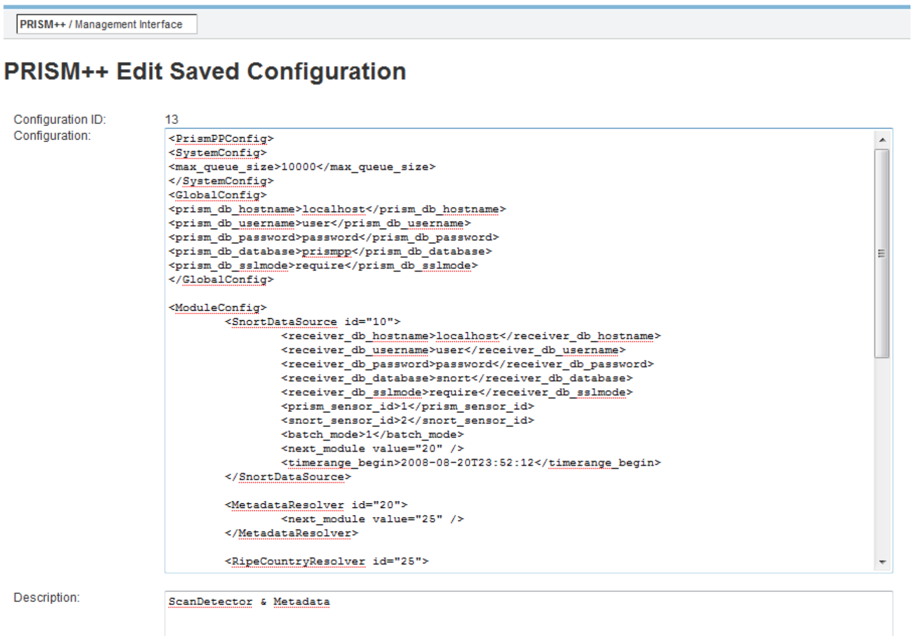
Add new configuration ...

PRISM++ © 2008 INF7, FAU, Licensed under the GNU GPL-2

Figure 2.5.: Management of Prism++ configurations

The edit screen for configurations is displayed in figure 2.6. Each configuration contains four elements:

- `Configuration ID`: Unique ID number (automatically assigned)
- `Configuration`: XML based configuration for the correlation engine
- `Description`: May contain an additional description for the configuration
- `Name`: Configuration's name



PRISM++ Edit Saved Configuration

Configuration ID: 13

Configuration:

```
<PrismPPConfig>
<SystemConfig>
<max_queue_size>10000</max_queue_size>
</SystemConfig>
<GlobalConfig>
<prism_db_hostname>localhost</prism_db_hostname>
<prism_db_username>user</prism_db_username>
<prism_db_password>password</prism_db_password>
<prism_db_database>prismpp</prism_db_database>
<prism_db_sslmode>require</prism_db_sslmode>
</GlobalConfig>

<ModuleConfig>
        <SnortDataSource id="10">
                <receiver_db_hostname>localhost</receiver_db_hostname>
                <receiver_db_username>user</receiver_db_username>
                <receiver_db_password>password</receiver_db_password>
                <receiver_db_database>snort</receiver_db_database>
                <receiver_db_sslmode>require</receiver_db_sslmode>
                <prism_sensor_id>1</prism_sensor_id>
                <snort_sensor_id>2</snort_sensor_id>
                <batch_mode>1</batch_mode>
                <next_module value="20" />
                <timerange_begin>2008-08-20T23:52:12</timerange_begin>
        </SnortDataSource>

        <MetadataResolver id="20">
                <next_module value="25" />
        </MetadataResolver>

        <RipeCountryResolver id="25">
```

Description: ScanDetector & Metadata

Figure 2.6.: Editing a Prism++ configuration

Session Management

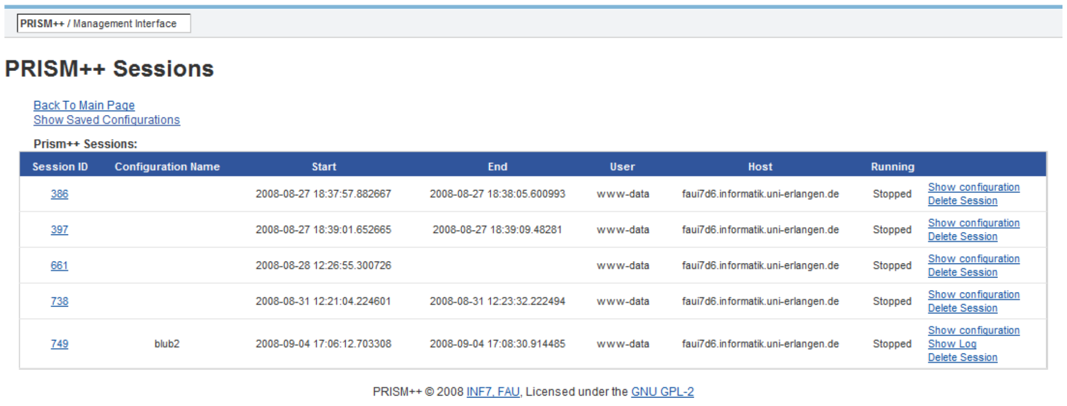
By clicking on link ‘Start new Session’ inside the configuration management screen, a new session may be started. Then the application starts the Python interpreter which starts the correlation engine with the assigned configuration. All active and inactive sessions are listed on the ‘Prism++ Session’ page (Figure 2.7). Active sessions may be aborted by clicking on link ‘Stop’. If the correlation engine needs to be aborted, the web interface may lag for 10 seconds, as it checks successful termination of the engine. Log files of the correlation engine are also displayed in this page. Internal cache management sometimes may cause lags between time of entry logs and display within the web interface.

2.2.2. Analysis of collected events

The web interface allows to visualize found events. Browsing to multiple pages offers the opportunity to manage high information quantities. Usage of the web interface is designed to be simple for analysists: it offers multiple views for statistical analysis of events, filtered and sorted display and personalized display of event lists.

Event table display

Results of the correlation engine are shown in page ‘Event Tables’ by the web interface. Depending on the navigation in the interface, either all tables are shown (by clicking on the link in the start screen), or only session-specific tables (by clicking on the links inside the Prism++ session page).



PRISM++ / Management Interface

PRISM++ Sessions

Back To Main Page
Show Saved Configurations

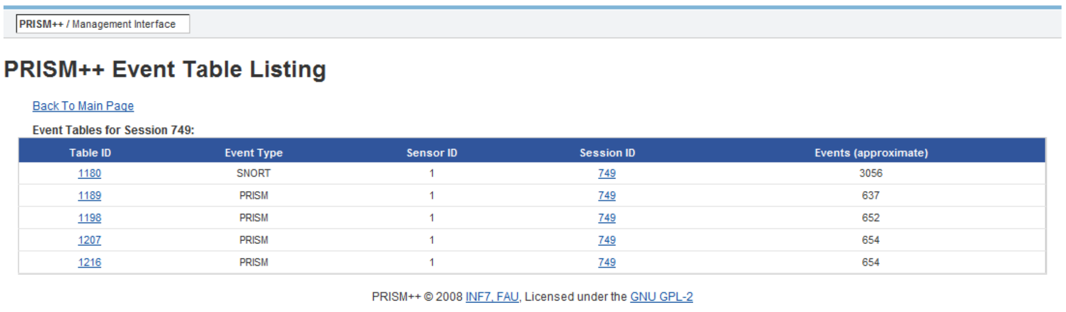
Prism++ Sessions:

Session ID	Configuration Name	Start	End	User	Host	Running	
386		2008-08-27 18:37:57.882667	2008-08-27 18:38:05.600993	www-data	faui7d6.informatik.uni-erlangen.de	Stopped	Show configuration Delete Session
397		2008-08-27 18:39:01.652665	2008-08-27 18:39:09.48281	www-data	faui7d6.informatik.uni-erlangen.de	Stopped	Show configuration Delete Session
661		2008-08-28 12:26:55.300726		www-data	faui7d6.informatik.uni-erlangen.de	Stopped	Show configuration Delete Session
738		2008-08-31 12:21:04.224601	2008-08-31 12:23:32.222494	www-data	faui7d6.informatik.uni-erlangen.de	Stopped	Show configuration Delete Session
749	blub2	2008-09-04 17:06:12.703308	2008-09-04 17:08:30.914485	www-data	faui7d6.informatik.uni-erlangen.de	Stopped	Show configuration Show Log Delete Session

PRISM++ © 2008 INF7, FAU, Licensed under the GNU GPL-2

Figure 2.7.: Active and stopped correlation engine sessions. Clicking on Session IDs lead to the tables containing correlated events

Exemplary, Figure 2.8 shows all tables contained in a session. Here events were received from Snort, correlated and the events were stored in four Prism++ tables.



PRISM++ / Management Interface

PRISM++ Event Table Listing

Back To Main Page

Event Tables for Session 749:

Table ID	Event Type	Sensor ID	Session ID	Events (approximate)
1180	SNORT	1	749	3056
1189	PRISM	1	749	637
1198	PRISM	1	749	652
1207	PRISM	1	749	654
1216	PRISM	1	749	654

PRISM++ © 2008 INF7, FAU, Licensed under the GNU GPL-2

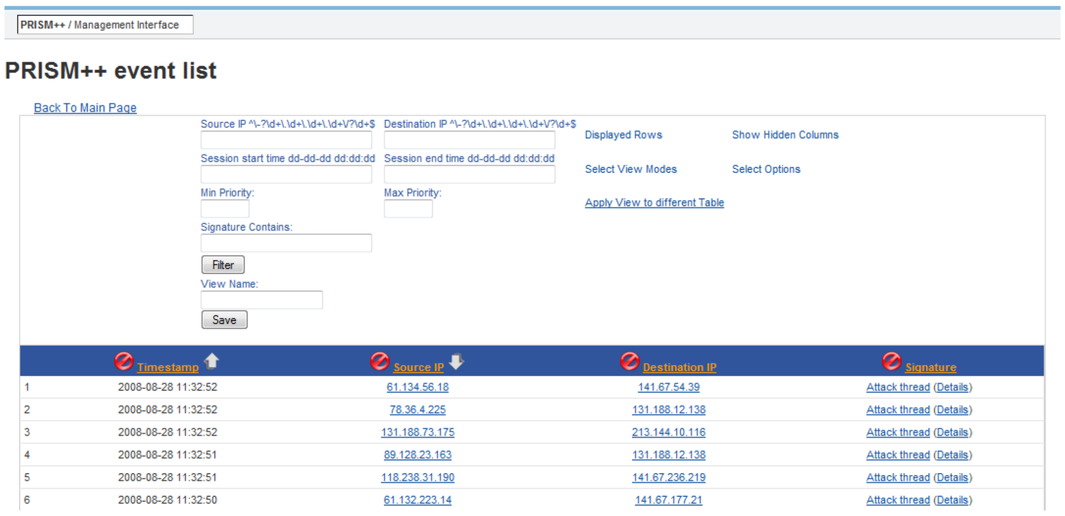
Figure 2.8.: Listing of event tables in Prism++

Event listing display

Each entry in the 'Event Tables' listing links to the corresponding pages displaying the event listings. In these pages, occuring events and meta events are displayed with parameters like time stamp, signature, source IP and destination IP.

Figure 2.9 shows an example, where a listing of Prism++ events is displayed which contain correlated data. Multiple filters and options can be selected at the top of the page.

Meta events are aggregatd events of type PRISM. The aggregated events were grouped by the correlator and stored in the event table. Meta events may represent a vertical or horizontal port scan, or event signatures from identical source adresses and destination addresses. Detailed information about the aggregated events can be displayed by clicking on the link called 'Details'.



PRISM++ / Management Interface

PRISM++ event list

Back To Main Page

Source IP ^\-?\d+\.\d+\.\d+\.\d+\/?\d+$
Destination IP ^\-?\d+\.\d+\.\d+\.\d+\/?\d+$
Session start time dd-dd-dd dd:dd:dd
Session end time dd-dd-dd dd:dd:dd
Min Priority:
Max Priority:
Signature Contains:

	Timestamp	Source IP	Destination IP	Signature
1	2008-08-28 11:32:52	61.134.56.18	141.67.54.39	Attack thread (Details)
2	2008-08-28 11:32:52	78.36.4.225	131.188.12.138	Attack thread (Details)
3	2008-08-28 11:32:52	131.188.73.175	213.144.10.116	Attack thread (Details)
4	2008-08-28 11:32:51	89.128.23.163	131.188.12.138	Attack thread (Details)
5	2008-08-28 11:32:51	118.238.31.190	141.67.236.219	Attack thread (Details)
6	2008-08-28 11:32:50	61.132.223.14	141.67.177.21	Attack thread (Details)

Figure 2.9.: PRISM++ event display with filter and option fields

Filter

The event listing page offers several possibilities for filtering events and allows the user to adjust the data view.

Filter for source or destination IP addresses can be set by editing the corresponding fields, but also by clicking on IP addresses in the event lists. Filters for signatures, priorities and time stamps may be set accordingly.

The IP addresses are checked for syntactic correctness after input. If the check fails, a message is displayed at the top of the page. It is possible to enter multiple IP addresses and to negate IPs using the prefix '-'. If we would enter '-131.13.1.74' in the field filtering the source addresses, all events not originating from IP 131.13.1.74 would be displayed. Subnet masks can also be specified, like '218.202.22.35/24'. Filters for subnets are implemented for masks going from /2 to /31. Negated IPs, subnet masks and regular IPs may be combined in the filters.

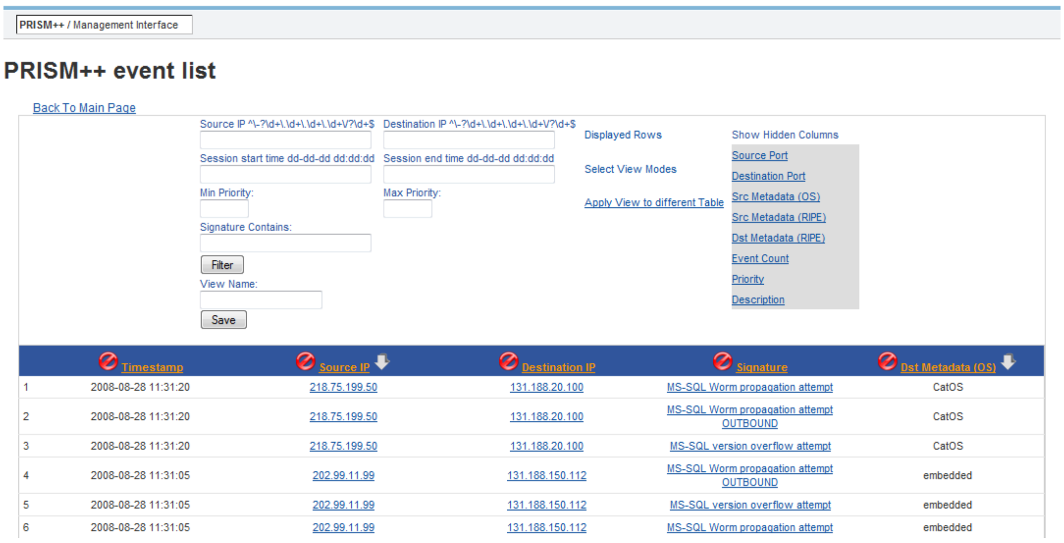
Options

Some additional options can also be used in the event listing page:

- Displayed Rows: Allows the adjustment of the number of events displayed in one page. Elements for navigating multiple pages are added at the bottom of the page if necessary.
- Select View Modes: Event lists may be statistically aggregated and displayed, therefore the interface provides the views 'Top signatures', 'Top source IPs', 'Top destination IPs' and 'Top IP pairs'.

- Select Options: By selecting ‘Live mode’, the user may switch the web interface to the Live mode, that enables the web interface automatic page refreshes. This is useful for viewing tables that are being filled by the correlator at the same time. In this menu, filters may also be removed

- Show Hidden Columns: Displayed columns may be adjusted: non-visible columns that are available in the current view are listed in this menu. If a column is selected, it is added to the event table. Figure 2.10 shows an example of this menu. Current sorting preferences are displayed at the side of the column names at the top of the event table. By clicking on the column names, sorting preferences may be changed.


PRISM++ / Management Interface

PRISM++ event list

Back To Main Page

Source IP ^\-?\d+\.\d+\.\d+\.\d+\/?\d+$
Destination IP ^\-?\d+\.\d+\.\d+\.\d+\/?\d+$
Session start time dd-dd-dd dd:dd:dd
Session end time dd-dd-dd dd:dd:dd
Min Priority:
Max Priority:
Signature Contains:
View Name:

Displayed Rows
Select View Modes
Apply View to different Table

Show Hidden Columns
Source Port
Destination Port
Src Metadata (OS)
Src Metadata (RIPE)
Dst Metadata (RIPE)
Event Count
Priority
Description

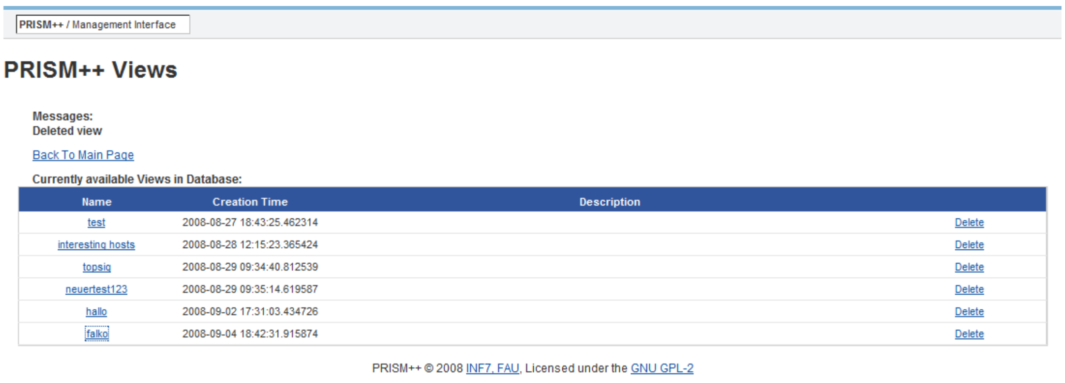
	Timestamp	Source IP	Destination IP	Signature	Dst Metadata (OS)
1	2008-08-28 11:31:20	218.75.199.50	131.188.20.100	MS-SQL Worm propagation attempt	CatOS
2	2008-08-28 11:31:20	218.75.199.50	131.188.20.100	MS-SQL Worm propagation attempt OUTBOUND	CatOS
3	2008-08-28 11:31:20	218.75.199.50	131.188.20.100	MS-SQL version overflow attempt	CatOS
4	2008-08-28 11:31:05	202.99.11.99	131.188.150.112	MS-SQL Worm propagation attempt OUTBOUND	embedded
5	2008-08-28 11:31:05	202.99.11.99	131.188.150.112	MS-SQL version overflow attempt	embedded
6	2008-08-28 11:31:05	202.99.11.99	131.188.150.112	MS-SQL Worm propagation attempt	embedded

Figure 2.10.: Hidden columns may be added to the table again

Views

So-called ‘views’ may be saved by setting a name inside the event listing page for the current view and clicking on the button ‘Save’. Then the interface saves all current view settings inside the database. All saved views can be accessed by the link ‘Show Saved Views’ in the web interfaces’ start screen. The resulting page is displayed in Figure 2.11. All saved views are listed in this table and by selecting the views, the stored views can be displayed again. Note that only display parameters and source event table are stored in the views. If the displayed table’s contents have changed since, the view will display different data.

The current display of a page may also be applied to other tables by clicking on link ‘Apply View to different Table’.


PRISM++ / Management Interface

PRISM++ Views

Messages:
Deleted view

Back To Main Page

Currently available Views in Database:

Name	Creation Time	Description	
test	2008-08-27 18:43:25.462314		Delete
interesting hosts	2008-08-28 12:15:23.365424		Delete
topsig	2008-08-29 09:34:40.812539		Delete
neuertest123	2008-08-29 09:35:14.619587		Delete
hallo	2008-09-02 17:31:03.434726		Delete
falko	2008-09-04 18:42:31.915874		Delete

PRISM++ © 2008 INF7, FAU, Licensed under the GNU GPL-2

Figure 2.11.: Selection of saved views

3. Installation

This chapter describes the installation steps necessary for installation and execution of the correlation engine and the web interface.

3.1. Correlation engine

1. The following applications and libraries are needed for the correlation engine:
   - python (>= 2.5)
   - postgresql (>= 8.3)
   - python-4suite-xml
   - python-pydot
   - python-dns
   - python-ipy
   - python-pygresql
2. setup PostgreSQL by creating an appropriate user for the correlation engine
3. setup tables in PostgreSQL by using script `prismpp.sql`
4. create configuration file, examples can be found in directory `correlator/config`
5. go to directory `correlator`
6. start engine with command `python start.py <configuration file>`

3.2. Webinterface

1. The following applications and libraries are needed for the web interface:
   - python (>= 2.5)
   - postgresql (>= 8.3)
   - apache2
   - libapache-mod-python
   - python-4suite-xml
   - python-pydot
   - graphviz

- rrdtool
- python-cheetah
- python-dns
- python-ipy
- python-pygresql
- an installed version of Prism++ correlator including database setup

2. copy configuration file `webinterface/prismpp.conf.template` to `webinterface/prismpp.conf` and
   - insert DB credentials of already created Prism++ database
   - update command line for executing the Prism++ correlation engine
3. execute Makefile in directory `webinterface/removesigmasks` with command `make`
4. copy files and subdirectories in directory `webinterface` to a location accessible by Apache
5. use file `webinterface/apache-sample.conf.template` as template for Apache's configuration
6. launch browser and access URL configured in Apache

4. Structure and Configuration

4.1. Correlation engine

Configuration

The configuration of the correlation engine is controlled by a XML file, which contains both global parameters relevant for all modules and a specific section for each module instance.

While the global parameters are accessible by all components of the correlation engine and, therefore, contain information such as the credentials for accessing the PRISM++ database, the module configuration contains all modules used in the correlator including their specific configuration and information on their interconnections.

```
<PrismPPConfig>
<GlobalConfig>
 [...]
</GlobalConfig>
<ModuleConfig>
 <SnortDataSource id="1">
  [...]
  <next_module value="2" />          (1)
  <next_module value="3" />          (2)
 </SnortDataSource>
 <AttackThreadDetector id="2">
  [...]
  <next_module value="4" />          (3)
 </AttackThreadDetector>
 <ScanDetector id="3">
  [...]
  <next_module value="4" />          (4)
 </ScanDetector>
 <AlertDBWriter id="4">
  [...]
 </AlertDBWriter>
</ModuleConfig>
</PrismPPConfig>
```

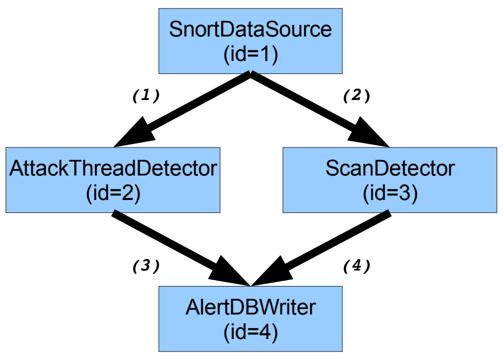
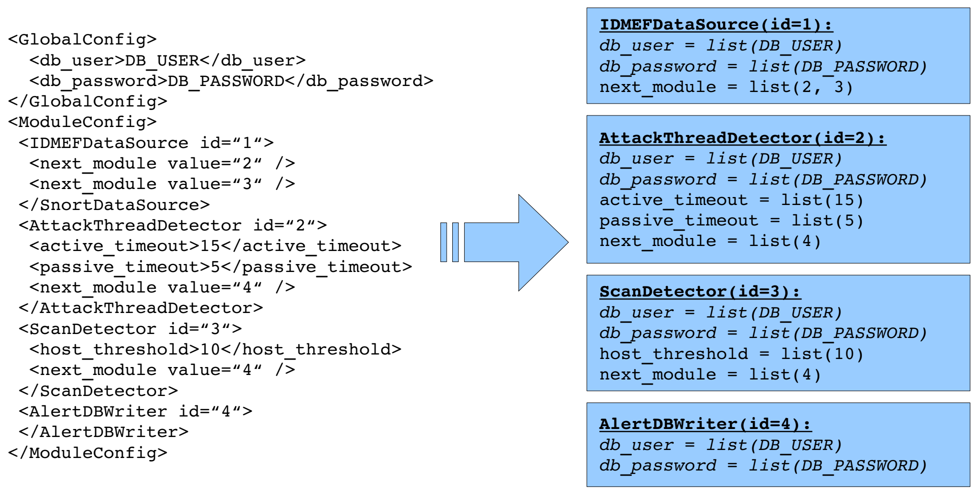


Figure 4.1.: PRISM++ configuration and module interconnections

Basically interconnections between modules are set up by using the special parameter `next_module`, which contains the ID of a module to which all outgoing events should be forwarded. The parallel forwarding of all events to multiple successors is realized by simply assigning multiple module IDs to this parameter.

A simple PRISM++ configuration and the results of the respective statements on the interconnection of the modules is outlined in Figure 4.1.

The simple example in Figure 4.2 demonstrates the inheritance of global parameters and the internal data structure provided to each module after parsing the XML file.



```
<GlobalConfig>
  <db_user>DB_USER</db_user>
  <db_password>DB_PASSWORD</db_password>
</GlobalConfig>
<ModuleConfig>
 <IDMEFDataSource id="1">
  <next_module value="2" />
  <next_module value="3" />
 </SnortDataSource>
 <AttackThreadDetector id="2">
  <active_timeout>15</active_timeout>
  <passive_timeout>5</passive_timeout>
  <next_module value="4" />
 </AttackThreadDetector>
 <ScanDetector id="3">
  <host_threshold>10</host_threshold>
  <next_module value="4" />
 </ScanDetector>
 <AlertDBWriter id="4">
 </AlertDBWriter>
</ModuleConfig>
```

```
IDMEFDataSource(id=1):
db_user = list(DB_USER)
db_password = list(DB_PASSWORD)
next_module = list(2, 3)

AttackThreadDetector(id=2):
db_user = list(DB_USER)
db_password = list(DB_PASSWORD)
active_timeout = list(15)
passive_timeout = list(5)
next_module = list(4)

ScanDetector(id=3):
db_user = list(DB_USER)
db_password = list(DB_PASSWORD)
host_threshold = list(10)
next_module = list(4)

AlertDBWriter(id=4):
db_user = list(DB_USER)
db_password = list(DB_PASSWORD)
```

Figure 4.2.: Data structure for module configuration

The parameters set in the `GlobalConfig` section are inserted into all module-specific configurations and thereby provide a convenient way of defining parameters such as database credentials or the log level globally. To identify them in the figure, they are written in italic letters.

Another important property is the representation of all parameters by lists, regardless of their cardinality. This simplifies the parsing process and ensures maximum flexibility.

4.1.1. Database Schema

The database schema of the PRISM++ correlation engine is outlined in Figure 4.3.

4.1.2. Configuration

An overview of the configuration parameters for all correlator modules and for specific modules is presented in the following section.

Global Parameters

Parameter	Description
prism_db_hostname	Hostname of the PRISM++ database server
prism_db_database	Name of the PRISM++ database
prism_db_username	Username for the PRISM++ database
prism_db_password	Password for the PRISM++ database
log_level	Log level for the logging subsystem

Table 4.1.: Global parameters of the PRISM++ correlation engine

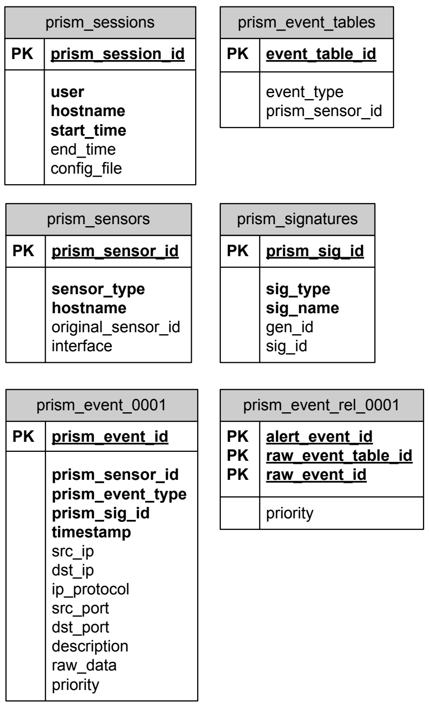


Figure 4.3.: PRISM++ database schema

Global parameters are propagated to all correlator modules and, therefore, provide an easy way to specify the database credentials for the PRISM++ database, which are used by various components. Table 4.1 shows a listing of currently used global parameters.

Data Sources

The main task of a correlation engine is the analysis of incoming events. Because a variety of different intrusion detection sensors exist and most of them are unfortunately not supporting standards like IDMEF yet, the design of all components involved in the reception of events should be kept simple to facilitate the implementation of support for new data sources.

As an important goal of the PRISM++ correlation engine is reproducibility and the support for both operation in real time and later in-depth analysis of the same data, all events arriving at the correlation engine must be archived for later reference. Furthermore, the process of normalization must be conducted on each incoming event.

Although the architecture of a data source depicted in Figure 4.4 is not mandatory, it is used by the two modules implemented so far.

It consists of an *event receiver*, which normalizes all incoming events by converting them to their internal representation, and a *database writer*, whose task is to store all received events in the PRISM++ database.

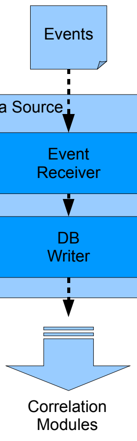


Figure 4.4.: Architecture of a PRISM++ data source

The method used by the event receiver for initializing the internal representation of an event is usually implemented by the respective subclass for the received event type.

If no archiving of the received events would be required, their normalized representation could be directly forwarded to any other correlator module. But in addition to the disadvantage that no subsequent analysis of the received events would be possible, the current mechanism for storing alerts described in section 4.1.2 relies on a persistent storage of the correlated events.

Therefore, this configuration is currently not supported and a data source must contain an event writer. Its event processing consists solely of writing the events to a database and assigning the resulting reference information to the respective attributes. Each event received by a PRISM++ data source is thereby uniquely identified by the table in which the event is stored and the `prism_event_id`, which is unique in each table.

After outlining the fundamental architecture of a data source in PRISM++, we now present the two currently implemented data sources.

SnortDataSource There are various methods for obtaining generated alerts from Snort instances, but most of them either involve parsing text files or even binary files in the case of the unified log format. The Prelude output plugin would require an additional implementation based on the libraries for its protocol as well.

The output plugin for databases, however, provides easy access to all data without additional implementation effort.

Based on these facts, the *SnortDataSource* module, a PRISM++ data source for receiving Snort alerts by accessing a PostgreSQL database, was implemented.

According to the proposed architecture for data sources already described, the SnortDataSource consists of a *SnortDBReader* instance, which fetches the alerts from the Snort database, and a *SnortEventWriter* for storing the received events in the PRISM++ database.

Parameter	Description
receiver_db_hostname	Hostname of Snort database
receiver_db_username	Username of Snort database
receiver_db_password	Password for database access
receiver_db_sslmode	Used SSL mode for database connection (usually 'require')
prism_sensor_id	Assigned sensor ID for this Snort instance inside Prism DB
snort_sensor_id	Sensor ID of Snort instance, which events should be used as input (most of the time 1)
batch_mode	Set to '1', if Prism++ stops if end of events is reached. Set to '0' if Prism++ should continue looking for events
timerange_begin	Specifies time, from which on events should be read from database. Example '2008-08-20T23:00:00'

Table 4.2.: Parameters for module SnortDataSource

The SnortDBReader exists in two versions, the *SnortLiveDBReader* and the *Snort-BatchDBReader*.

While the first module periodically fetches new alerts from Snort's database and, therefore, is suited for real-time operation, the SnortBatchDBReader reads all Snort alerts from the database specified in the module's configuration with a single query and terminates afterwards. Its main use case is the import of Snort alerts into the PRISM++ database in collaboration with a SnortEventWriter.

To determine the current position in the database and to assure that every record is only fetched once, Snort's event ID is used and each SnortDBReader instance stores the next ID expected internally.

Consequently, this has to be detected by the initialization method of the SnortEvent class, which raises an exception indicating this situation. The receiver module, in turn, stops its processing of the remaining events and waits for the configured query interval. In the next iteration, it should then be able to initialize the event correctly.

Another issue results from the fact that Snort sometimes writes alerts to the database in an order which is not consistent with respect to the timestamp of the events. As we are using the event ID to avoid duplicate or missing events and thus receive all events in the order of their ID, the temporal order of the event stream would be violated.

After the events are received and normalized by the SnortDBReader modules, the resulting SnortEvent objects are forwarded to the SnortEventWriter module, which stores each SnortEvent in the PRISM++ database and thereby makes it persistent.

On startup, a SnortDataSource instance requests a new event table from the Table-Manager before processing any events. All events subsequently received by this module are then written to this table.

To provide a unique identification for Snort signatures across different Snort instances and databases, for each received event, a `prism_sig_id` for this signature is looked up and assigned to the event. If the Snort signature is not yet known to PRISM++, a new record for this signature is created in the PRISM++ database and the resulting ID is used. This process relies on unique signature IDs coming from Snort with a specified sensor ID. If Snort's internal ID assignment does not produce unique signature IDs any more (like when completely deleting Snort's internal database structure), Prism++ will not be able to determine the correct signature name by only a signature ID. Users must make sure that this case does not happen.

After the event has been written to the database, it is uniquely identified by the respective table name and the ID of the event in this table. Finally the event is forwarded to subsequent correlator modules.

IDMEFDataSource The module *IDMEFDataSource* is capable of receiving IDMEF messages via HTTP on port 8888. This data source also follows the architecture presented in Figure 4.4 and consists of an *IDMEFReceiver* and an *IDMEFEventWriter*.

IDMEF is a very comprehensive and complex message format for describing intrusion detection alerts in detail. Consequently, the effort for implementing a complete parser for its data structure presented in section IDMEF should not be underestimated.

For our purposes, a support for the basic attributes of an IDMEF message such as the timestamp, source and destination IP addresses and the classification of the event is sufficient.

Similar to the SnortEventWriter, a IDMEFEventWriter receives IDMEFEvent objects from the IDMEFReceiver module and writes them to the database for later reference.

The table for these events is created by requesting it from the TableManager.

To provide a unique identification of IDMEF messages from different sensors, according to the IDMEF RFC, the `ident` attribute of the alert's classification should contain a unique identifier for each alert classification [DCF07]. Unfortunately this attribute is marked as optional and only the `text` attribute is mandatory.

For uniquely identifying the signature used within PRISM++ with the help of the SignatureManager, we have to use a workaround for this issue and, therefore, use the `text` attribute for unique signature identification, if the alert contains no value for the identification attribute.

Finally, the IDMEF events are written to the respective event table, the attributes for uniquely identifying the event, the event table's name and the respective event ID, are assigned and the IDMEFEvent object is forwarded to the next correlator module.

PrismEventReader An important feature of the PRISM++ correlation engine is the batch analysis of already received events. This enables security analysts and researchers to analyze the events in detail or to search for patterns and anomalies with different configurations of the correlation engine or its modules.

The functionality of retrieving archived events from the PRISM++ database is realized by the *PrismEventReader* class, which fetches events from one or more tables, forwards them and thereby simulates the arrival of the respective events in real time.

The artificial field assigning the table ID to the field `table_id` is required to represent the location of each event in the database and for the correct initialization of the event's `prism_table_id` attribute.

For each event resulting from the query, the event type is determined by the field `prism_event_type` and an according instance for its representation is created.

Correlation Modules

After presenting the common subsystems and the data sources for receiving or reconstructing the flow of intrusion detection events, the actual correlator modules are presented.

Each of these modules analyzes incoming events using a specific algorithm, tries to identify patterns and generates alerts if it is successful. Another purpose of such modules is the aggregation and prioritization of events to reduce the number of alerts presented to an operator.

To facilitate the actual implementation of the modules, some common data structures useful for at least some of these algorithms were created.

InclusiveEventFilter and ExclusiveEventFilter To analyze events related to specific IP addresses or representing the detection of a specific signature, two filter modules were implemented.

Parameter	Description
prism_sig_id	PRISM's internal ID for a signature
src_ip_filter	Matching source IP addresses or networks
dst_ip_filter	Matching destination IP addresses or networks
ip_filter	Matching IP addresses or networks source or destination
priority_from_filter	Minimum priority value for events
priority_to_filter	Maximum priority value for events

Table 4.3.: Parameters for modules InclusiveEventFilter and ExclusiveEventFilter

As all parameters are implicitly realized as lists, which was already stated in section 4.1, matching multiple values for one parameter type is possible by simply specifying the parameter multiple times in the module's configuration.

The *InclusiveEventFilter* forwards only events matching its filter conditions to subsequent correlator modules, while the *ExclusiveEventFilter* removes all matching events and forwards only events, which do not match its filters.

Parameter	Description
active_timeout	Active timeout in seconds that specifies the maximum time of entries to be buffered
passive_timeout	Passive timeout in seconds that specifies the minimum time of entries to be buffered. They are exported as soon as passive_timeout seconds no event was added to the entry.

Table 4.4.: Parameters for module AttackThreadDetector

AttackThreadDetector The idea and the name for this module's correlation algorithm, *attack thread reconstruction*, was taken from Fredrik Valeur's PhD thesis [Val06] and the article published by him and his colleagues at the University of California, Santa Barbara

[VVKK04]. It represents an aggregation of all events having the same originator and the same target in a time period up to a configurable timeout.

Consequently this algorithm only supports events having a single source and destination IP address and if an event violating this constraint is received, an exception is raised.

To temporarily store the events, the *AttackThreadDetector* uses both common data structures described previously, the EventStorage and the MetaEventStorage class. For each incoming event, a cache of already detected attack threads, which is implemented using a MetaEventStorage instance, is queried for a matching event. If this search is successful, the incoming event is merged with the existing meta event representing this attack thread. Otherwise, the cache of recently received event objects realized by an EventStorage instance, is searched for events with source and destination IP addresses identical to the attributes of the incoming event. If a corresponding event is found, a new attack thread instance, including both the event just received and the one from the cache, is created and the representation of the resulting meta event is stored by the MetaEventStorage instance.

To control the behavior of this algorithm, two parameters are available. While the `active_timeout` has an influence on the maximum length of an attack thread, regardless of any ongoing activity, the `passive_timeout` represents the time period after which an attack thread supposedly ended because no further activity occurred.

Consequently, the active timeout must be higher than the passive timeout to guarantee correct functionality. This constraint is ensured by the module's implementation and an exception is raised if it is violated.

Both parameters have an influence on the time period, for which events may be kept in this correlation module. While the passive timeout has a mostly irrelevant influence on the time an event remains in the AttackThreadDetector module, the active timeout represents the maximum time any event is kept in this module. This has to be considered, if subsequent correlator modules are using caches or timeouts as well, because the temporal order of the events leaving this module may be broken depending on the detected attack threads.

Parameter	**Description**
active_timeout	Active timeout in seconds that specifies the maximum time of entries to be buffered
passive_timeout	Passive timeout in seconds that specifies the minimum time of entries to be buffered. They are exported as soon as passive_timeout seconds no event was added to the entry.
host_threshold	Multiple entries are aggregated to a 'Scan' metaevent, as soon as at least this number of entries would be aggregated.

Table 4.5.: Parameters for module ScanDetector

ScanDetector To detect attacks from one originator to multiple destinations, which are often some form of a scan, we implemented a *ScanDetector*. The idea and the algorithm for this module was also taken from Valeur's PhD thesis, who named the component containing this algorithm *attack focus recognition* and calls the specific pattern of activity often originating from scanning activity the *one2many* scenario [Val06]. It tries to detect source hosts which cause events directed to a large number of targets.

Because of the algorithm currently used for this module, it supports no events with multiple source or destination addresses and raises an exception if such an event is received.

For each received event, it is first checked if a scan coming from its source IP was already detected. In this case, the event is added to the list of events associated with this scan.

The number of destination hosts required to classify the activity as a scan is controlled by the parameter `host_threshold`. To control the temporal aspects of the algorithm, the parameters for active and passive timeout already described previously can be used.

The potential impact on the temporal order of the forwarded events must be considered for this module as well.

Parameter	**Description**
active_timeout	Active timeout in seconds that specifies the maximum time of entries to be buffered
passive_timeout	Passive timeout in seconds that specifies the minimum time of entries to be buffered. They are exported as soon as passive_timeout seconds no event was added to the entry.
host_threshold	Multiple entries are aggregated to a 'DistributedAttack' metaevent, as soon as at least this number of entries would be aggregated.

Table 4.6.: Parameters for module ScanDetector

DistributedAttackDetector To detect attacks coming from various hosts and focusing on one specific target, a *DistributedAttackDetector*, which is essentially a ScanDetector detecting activity in the contrary direction, was developed as well. This kind of algorithm is especially useful to detect *distributed denial-of-service(DDoS) attacks*[1] and to identify the botnets[2] often used for such activities.

To implement a correlation algorithm for detecting this type of behavior, the source code for the ScanDetector module was copied and slightly modified to adapt it to its new task.

Thus, the fundamental architecture and design largely resembles the ScanDetector class. Because of the similar algorithm, it also raises an exception on the arrival of an event with multiple source or destination IP addresses.

The parameters supported by this module are identical to the ones offered by the ScanDetector.

[1]attacks on one target by a large number of hosts, usually used to cause overload on a system
[2]large networks of infected systems posing a serious threat

Parameter	Description
cache_timeout	Defines how long DNS hostnames are to be buffered to avoid multiple DNS requests
query_timeout	DNS query timeout

Table 4.7.: Parameters for module DnsResolver

DnsResolver The *DnsResolver* module tries to resolve the associated names for the source and destination IP addresses of each incoming event by querying the Domain Name System(DNS).

If the DNS lookup is successful, the hostnames for the source and destination IP addresses are written to the respective attributes `src_hostname` and `dst_hostname`. Otherwise, an empty string is assigned to the attribute.

To further improve the performance of the correlation module, a very simple cache for already retrieved mappings of IP addresses to hostnames was implemented. Obsolete entries in this cache are avoided by a timeout mechanism, which removes each entry after 600 seconds. Of course, this timeout may be changed as well by setting the `cache_timeout` parameter.

RipeCountryResolver Another potentially valuable information based on IP addresses is the location of the respective hosts. Although this information may be incorrect because of proxy servers or VPN systems, it provides at least some hints to the real location of the system.

The data for determining the country code for an IP address was extracted from the databases provided by the agencies allocating ranges of IP addresses to organizations or individuals. The location of these databases and some hints for processing them was found in the documentation of the *IP::Country*[3] Perl module.

By processing these databases with the help of the `ipcc_loader` script, which is also provided by the IP::Country module, a text file containing the network address, the size of the network and the two-letter country code for each currently assigned IP network is created.

This text file is then parsed by a simple Perl script and for each IP network, a record is inserted into the table `ip_to_country` outlined in Figure 4.5

The IP address of the network and its size are stored in the fields `net_address` and `net_size`, while the country code for the respective network is assigned to the `country` field. Although the two additional attributes `first_address` and `last_address` contain redundant information, they facilitate the database query.

The RipeCountryResolver module itself just looks up the corresponding entries for the source and destination IP addresses in the previously populated table and assigns the resulting two-letter country code to the attributes `src_country` and `dst_country`.

[3]http://search.cpan.org/ nwetters/IP-Country-2.24/

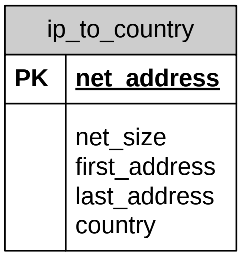


Figure 4.5.: Database table containing the country mapping for IP addresses

MetadataResolver The *MetadataResolver* module adds metadata to incoming events. For each event's source and destination IP, the module looks up the following metadata per host:

- operating system
- critflags (both data for flag names and assignment to IP addresses have to be inserted by user)
- open ports
- time since last reboot
- router hops to machine

All metadata is stored in tables inside Prism++ database. These tables are created using the default Prism++ database creation script. Several scripts in directory `metadata_scripts` allow easy insertion and modification of the contents of the metadata tables.

The module assumes that all metadata tables are located directly inside the default Prism++ database, so the global parameters for database access are used.

Prioritizing Module

After adding available metadata to events and correlation, i.e. aggregating and grouping them, all generated meta events need to be checked for relevancy. This is done by analyzing all events thoroughly, and, depending on their content, the meta events are assigned a priority. This task is performed by module *MetaEventPrioritizer*.

Priorities are configured by specifying Python code that analyzes all single events contained in meta events. Each single event is assigned a priority. Meta events, which may consist of multiple events, get the summed priority of all contained single events.

The Python code defined in parameter `prio_code` may access all fields in the events. These parameters are:

Parameter	Description
prio_code	Defines Python code for prioritizing all incoming events

Table 4.8.: Parameters for module DnsResolver

- *prio*: Priority of the event
- *ipsrc*: Source IP
- *ipdst*: Destination IP
- *ipproto*: Used Protocol
- *signame*: name of signature (usually Snort signature names)
- *srcport*: Source port (if available)
- *dstport*: Destination port (if available)
- *desc*: Description of event
- *srcmetaflags*: Critflags for source IP coming from metadata
- *dstmetaflags*: Critflags for destination IP coming from metadata

The value contained in variable `prio` is assigned as the final event's priority value. Section A.1 in the appendix shows a complete example for a Prism++ configuration including configuration of module *MetaEventPrioritizer*.

Output and Notification Modules

All correlator modules presented so far process incoming events according to their algorithm, filter or aggregate them and inject additional information. But the results produced by them have to be communicated to an operator or other systems. Additionally, they should be stored persistently to enable later in-depth analysis.

Currently two modules of this class are available, the *AlertDBWriter* and the *MailNotifier*.

AlertDBWriter The alert writer is a rather simple component which maps the events and their data to database fields and creates a record for each received event. Parameters are not needed for this module, as the default database is used as destination.

As we focused on simple interfaces throughout the design of the correlation engine and the adjoining database wherever possible, a shared database schema is used for raw events and for alerts.

The meta event itself is stored in the database by inserting it as an ordinary event with a special type. All aggregated events in its container are already stored in the database because of the functionality required by each data source implementation. Thus, for each aggregated event a record representing the association with its meta event is inserted.

Of course, some additional issues regarding the storage of meta events arise. For example, the issue of meta events using sets for various attributes is currently represented by setting the respective fields to *NULL* in the database. To determine the members of the set for a specific attribute, the process accessing the alerts needs to recursively traverse the events aggregated in the meta event.

MailNotifier To notify a security analyst or an operator in real time, we implemented a MailNotifier module which sends an email if a arriving event or meta event has a priority higher than a configurable threshold.

Although this rather simple form of real-time notification would need various improvements for operational usage, it demonstrates the principal possibilities offered by the real-time event processing of the correlation engine.

Modules for Debugging

EventSniffer A very useful tool for monitoring the event flow through the individual modules and their interconnections is the *EventSniffer* module.

Its output is directed to a file in the session directory of the currently active session of the correlation engine. By setting the `only_metaevents` parameter to 1, only meta events are logged.

EventInjector The *EventInjector* module is essentially just a template which demonstrates the ability to inject arbitrary events at any position in the module graph.

4.2. Webinterface

To provide easy access to the alerts generated by the PRISM++ correlation engine, a simple web-based user interface has been implemented, which is able to present the archived events received by sensors as well as the alerts generated by the correlation engine itself.

To demonstrate the ability to generate real-time notifications as well, we implemented a simple notification module, which sends an email to an operator, if an alert with a priority exceeding a configured threshold occurs.

A. Appendix

A.1. Example configuration

This section shows a complete configuration example for Prism++. Snort events are read from Snort's database in batch mode beginning at time 2008-08-20 at 23:52:12. The following procedure is performed for each incoming event:

1. Resolve available metadata
2. Get RIPE country for source and destination IP
3. Set Events' priority to 1
4. Write events to table
5. Perform correlation on events: detect attack threads, distributed attacks and scans
6. Perform priotization on meta events
7. Write events to table

```
<PrismPPConfig>
        <SystemConfig>
                <max_queue_size>10000</max_queue_size>
        </SystemConfig>
        <GlobalConfig>
                <prism_db_hostname>localhost</prism_db_hostname>
                <prism_db_username>prismpp</prism_db_username>
                <prism_db_password>password</prism_db_password>
                <prism_db_database>prismpp</prism_db_database>
                <prism_db_sslmode>require</prism_db_sslmode>
        </GlobalConfig>

        <ModuleConfig>
                <SnortDataSource id="10">
                        <receiver_db_hostname>localhost</↘
                           →receiver_db_hostname>
                        <receiver_db_username>snort</↘
                           →receiver_db_username>
                        <receiver_db_password>password</↘
                           →receiver_db_password>
                        <receiver_db_database>snort</↘
                           →receiver_db_database>
                        <receiver_db_sslmode>require</↘
                           →receiver_db_sslmode>
                        <prism_sensor_id>1</prism_sensor_id>
                        <snort_sensor_id>4</snort_sensor_id>
                        <batch_mode>1</batch_mode>
                        <next_module value="20" />
                        <timerange_begin>2008-08-20T23:52:12</↘
                            →timerange_begin>
                </SnortDataSource>

                <MetadataResolver id="20">
                        <next_module value="25" />
                </MetadataResolver>

                <RipeCountryResolver id="25">
                        <next_module value="27" />
                </RipeCountryResolver>

                <MetaEventPrioritizer id="27">
                        <prio_code>
# inside the prio_code tags, Python code must be specified. Correct
# indentation is essential for Python, so unfortunately the XML files
# indentation rules cannot be followed

# this module does not do anything: it just sets the current
# priority of events to 1
prio = 1
                        </prio_code>
                        <next_module value="28" />
                </MetaEventPrioritizer>

                <AlertDBWriter id="28">
                        <next_module value="31" />
                </AlertDBWriter>

                <AttackThreadDetector id="31">
                        <active_timeout>1800</active_timeout>
                        <passive_timeout>600</passive_timeout>
                        <next_module value="32" />
                </AttackThreadDetector>

                <DistributedAttackDetector id="32">
                        <active_timeout>1800</active_timeout>
                        <passive_timeout>600</passive_timeout>
                        <host_threshold>5</host_threshold>
                        <next_module value="34" />
                </DistributedAttackDetector>

                <ScanDetector id="34">
                        <active_timeout>1800</active_timeout>
                        <passive_timeout>600</passive_timeout>
                        <host_threshold>5</host_threshold>
                        <next_module value="36" />
                </ScanDetector>

                <MetaEventPrioritizer id="36">
```

```
                        <prio_code>
# prio: old priority of event
# ipsrc: source ip
# ipdst: destination ip
# ipproto: protocol
# signame: name of signature (mostly snort signature names)
# srcport: source port
# dstport: destination port
# desc: description
# srcmetaflags: meta critflags for source ip
# dstmetaflags: meta critflags for destination ip
prio = 1

# Prioritize keywords
keyword_prioritization = [
         ("COMMUNITY␣MISC␣BAD-SSL␣tcp␣detect", 0.01),
         ("COMMUNITY␣SIP␣TCP/IP␣message␣flooding␣directed␣to␣SIP␣proxy",↘
            → 0.02),
         ]
for (keyword, value) in keyword_prioritization:
  if signame.find(keyword) != -1:
    prio = value

if ipdst=='1.2.3.4' and signame in ("COMMUNITY␣WEB-ATTACKS␣GFI␣↘
    →MailSecurit
y␣Management␣Host␣Overflow␣Attempt␣Long␣Accept␣Parameter", "COMMUNITY␣↘
    →BOT␣GTBot
info␣command"):
  prio = 0.01

if 'server' in (srcmetaflags.keys() + dstmetaflags.keys()):
  prio *= 10
if 'noincoming' in dstmetaflags:
  prio = prio*1000+1000  # this should never happen!

if 'nfs' in srcmetaflags and 'nfs' in dstmetaflags and signame in ("↘
    →COMMUNITY␣MI
SC␣BAD-SSL␣tcp␣detect", "COMMUNITY␣SIP␣TCP/IP␣message␣flooding␣directed␣↘
    →to␣SIP␣p
roxy"):
  prio = 0.001
                        </prio_code>
                        <next_module value="40" />
                </MetaEventPrioritizer>

                <AlertDBWriter id="40">
                </AlertDBWriter>

        </ModuleConfig>
</PrismPPConfig>
```

Bibliography

[DCF07] H. Debar, D. Curry, and B. Feinstein. The Intrusion Detection Message Exchange Format (IDMEF). Technical Report RFC 4765, IETF, March 2007.

[KVHD06a] Jochen Kaiser, Alexander Vitzthum, Peter Holleczek, and Falko Dressler. Automated resolving of security incidents as a key mechanism to fight massive infections of malicious software. In *GI SIDAR International Conference on IT-Incident Management and IT-Forensics (IMF 2006)*, volume LNI P-97, pages 92–103, Stuttgart, Germany, October 2006. Springer.

[KVHD06b] Jochen Kaiser, Alexander Vitzthum, Peter Holleczek, and Falko Dressler. Ein Sicherheitsportal zur Selbstverwaltung und automatischen Bearbeitung von Sicherheitsvorfällen als Schlüsseltechnologie gegen Masseninfektionen. 1st GI SIG SIDAR Graduate Workshop on Reactive Security (SPRING), July 2006.

[Val06] Fredrik Valeur. *Real-Time Intrusion Detection Alert Correlation.* Ph.d. thesis, UC Santa Barbara, June 2006.

[VVKK04] Fredrik Valeur, Giovanni Vigna, Christopher Kemmerer, and Richard Krügel. A Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing*, 1(3):146–169, 2004.