Ref. Ares(2019)2905746 - 30/04/2019

Horizon 2020 Programme

Instrument: Innovation Action

Proactive Risk Management through Improved Cyber Situational Awareness

PROTECTIVE

PROACTIVE RISK MANAGEMENT

Start Date of Project: 2016-09-01

Duration: 36 months

D6.10 PROTECTIVE System v3

Deliverable Details	
Deliverable Number	**D6.10**
Revision Number	**E**
Author(s)	**GMV**
Due Date	**M28**
Delivered Date	**04/2019**
Reviewed by	**AIT, TUDA**
Dissemination Level	**PU**
Contact Person EC	**Alina-Maria Bercea**

The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under grant agreement no 700071.

Partner Roles

Contributing Partners	
1.	**GMV (deliverable responsible)**
2.	**AIT (reviewer)**

Revision History

Revision	By	Date	Changes
E	GMV, AIT, TUDA	30/04/2019	Final Version

Abbreviation's List

AC	Authorisation Code
API	Application Programming Interface
ASM	Asset State Management
CA	Context Awareness
CAFI	CA Fusion Inventory
CC	Client Credential
CSA	Cyber Situational Awareness
CVSS	Common Vulnerability Score System
DNS	Domain Name Server
EWMA	Exponential Weighted Moving Average
FMP	Future Misbehaviour Probability
GRU	Gated Recurrent Units
HTTPS	Hypertext Transfer Protocol Secure
IDEA	Intrusion Detection Extensible Alert
ISC	Information Sharing Compliance
JSON	Javascript Object Notation
KC	Keycloak
MAIR	Mission and Asset Information Repository
MAP	Meta-Alert Prioritization
MCDA	Multi-Criteria Decision Analysis
MIM	Mission Impact Model
NERD	Network Entity Reputation Database
NVD	National Vulnerability Database
Prot-Dash	PROTECTIVE-Dashboard
RNN	Recurrent Neural Networks
RA	Registration Authority
SOC	Security Operation Centre
TI	Threat Intelligence
UI	User Interface
URL	Uniform Resource Locator
VM	Virtual Machine
WSGI	Web Server Gateway Interface

Executive Summary

The PROTECTIVE System v3 is the third release of the integrated and validated PROTECTIVE system which will be used in the TestBed and during the Pilot 2.

It has been built mainly with the PROTECTIVE node based on the Warden (Warden, 2014) and Mentat (Mentat, 2017) systems of CESNET, a consortium partner. During the project, a wide variety of improvements have been performed, the default Mentat's User Interface (UI), HAWAT, has been replaced with a new UI, PROT-Dash, which provides a more customizable user's dashboard, as it allows addition and removal of widgets with custom graphs and perform custom queries to the database. Finally, the system includes the Context-Awareness (CA) software and Fusion inventory agent installers. Mentat's flow has been modified to have a correlation engine system through wso2da to produce meta-alerts and a new module for meta-alert prioritization. PROTECTIVE System also includes a set of connectors in order to ingest data into Warden and Mentat, which are Kippo, Dionaea, LaBrea, IntelMQ, Juniper SRX, SIEM Mcafee and Fortigate. Finally, an information sharing compliance module is available to ensure sensitive data is not shared.

The structure of the document is as follows. Chapter 1 overviews the document. Chapter 2 describes the ecosystems that can be created using the PROTECTIVE System. Chapter 3 describes the PROTECTIVE node in detail. Finally there is an Implementation chapter that describes the prerequisites to run the solution and where to find all the components.

Contents

The research leading to these results has received funding from the European Union's Horizon 2020 Research and Innovation Programme, under grant agreement no 700071.

List of Figures

List of Tables

1 Introduction

1.1 Overview

The focus of this document is to describe the current version of the PROTECTIVE System, v3.

The main functionalities of the PROTECTIVE System v3 are:

- Alert Processing Pipeline
- Alert Correlation
- Context-awareness (CA)
- Trust Module
- Meta-Alert Prioritisation
- Analytics
- Threat Intelligence (TI) Sharing
- Security
- User Interface

It has been built mainly with the PROTECTIVE node based on the Warden (Warden, 2014) and Mentat (Mentat, 2017) systems of CESNET.

Several connectors, Kippo, Dionaea, LaBrea, IntelMQ, Juniper SRX, SIEM Mcafee, MySQL database and Fortigate, have been developed to add to the system the ability to:

- Collect events from different sources
- Transform them into Intrusion Detection Extensible Alert (IDEA, 2017) format
- Push them into Ingestion subsystem

There are also instructions for creating new connectors.

In the following sections, all these components of the PROTECTIVE System are listed and described.

2 Threat Intelligence Ecosystem

PROTECTIVE has the ambition to develop a comprehensive solution to raise organisational cyber situational awareness (CSA) through:

1. Enhancement of security alert correlation and prioritisation
2. Linking of the relevance/criticality of an organizations assets to its business/mission
3. Establishment of a threat intelligence sharing community

It aims to do so by researching and developing a computing platform that will provide the CSA functions related to context awareness, threat awareness, as well as by developing the policies and mechanisms to enable threat intelligence sharing between CSIRT teams in a grander PROTECTIVE CTI community or ecosystem. Such an ecosystem is a federation of a number of PROTECTIVE nodes in different partner organisations which share information for the purposes of mutually improving identification and prevention and mitigation of threat events in their respective constituencies.

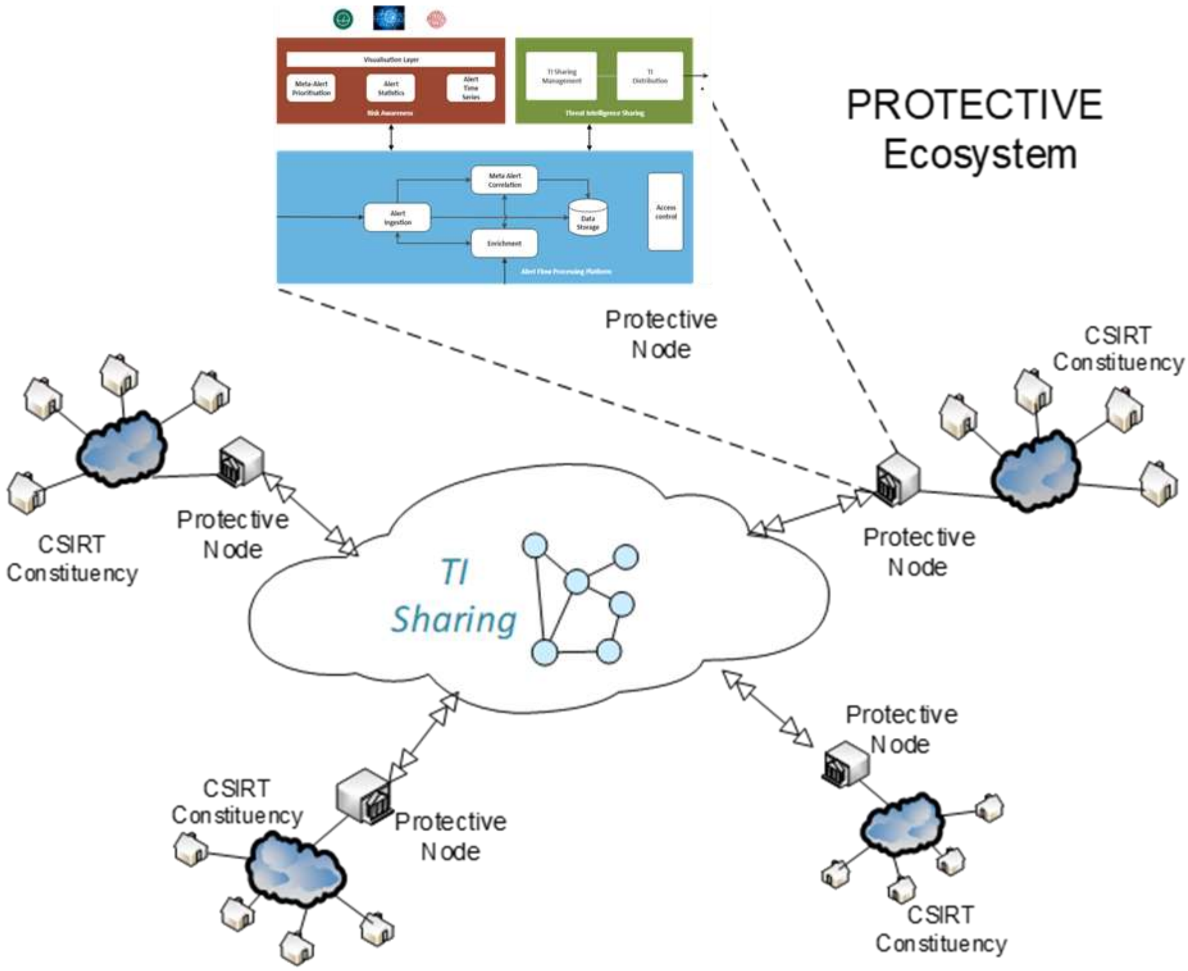

Figure 1: PROTECTIVE Ecosystem

This ecosystem is illustrated in Figure 1. This shows a number of CSIRT constituencies networks with their constituency members. Each network has (at least) one PROTECTIVE node which is used to fulfill the CSA goals above and also to route threat information to and from community partner PROTECTIVE node. This is depicted by the double ended arrows. The figure also shows a details inside an instance of a PROTECTIVE node.

PROTECTIVE nodes can be set up to share TI in a peer-to-peer architecture or a centralised architecture

2.1 PROTECTIVE P2P Architecture

The Peer-to-peer architecture represents separate communities, each with their own Warden Server and Mentat system instances – see Figure 2. A community has full control on what is received to its node and more importantly what is sent to other communities. This set-up is implemented by Mentat modules, which sort events according to policies and injects what is suitable into “foreign” Wardens.

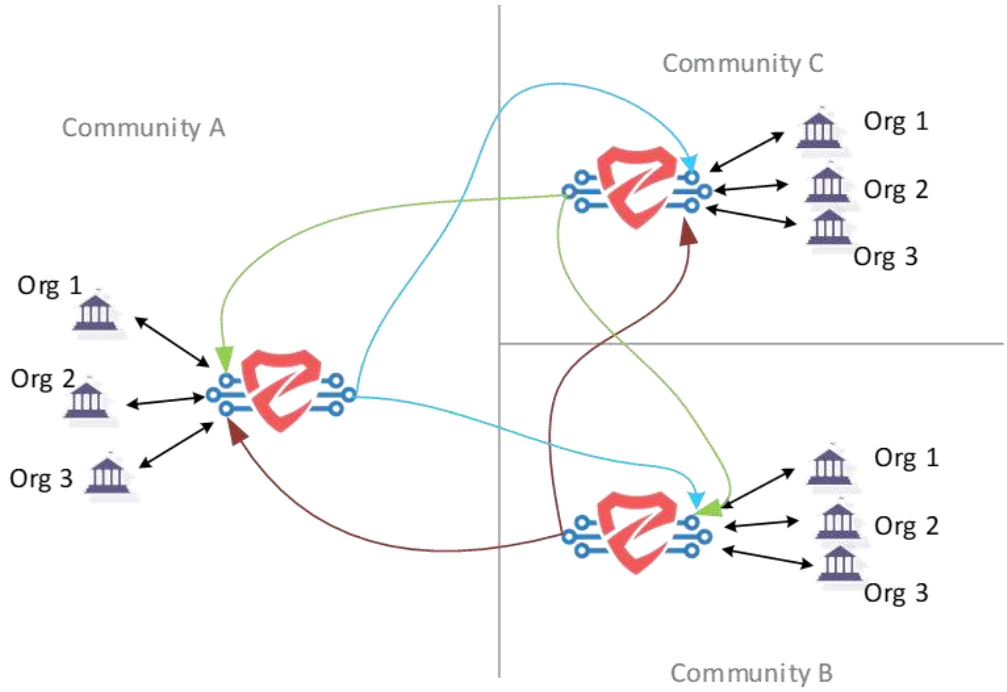

Figure 2: Peer-to-peer Architecture

2.2 PROTECTIVE Centralised Architecture

The PROTECTIVE Centralised Architecture represents separate communities, each with their own Warden and Mentat servers and a top-level (central) Warden Server – see Figure 3. The community has full control on what is received to its node and as well as what is sent to top-level (central) Warden Server. Top-level (central) Warden distributes everything (but only) what it gets from lower instances (nodes). This set-up is implemented by Mentat modules and an Information Sharing Compliance (ISC) module, which sort events according to policies and injects what is suitable into top-level (central) Warden Server.

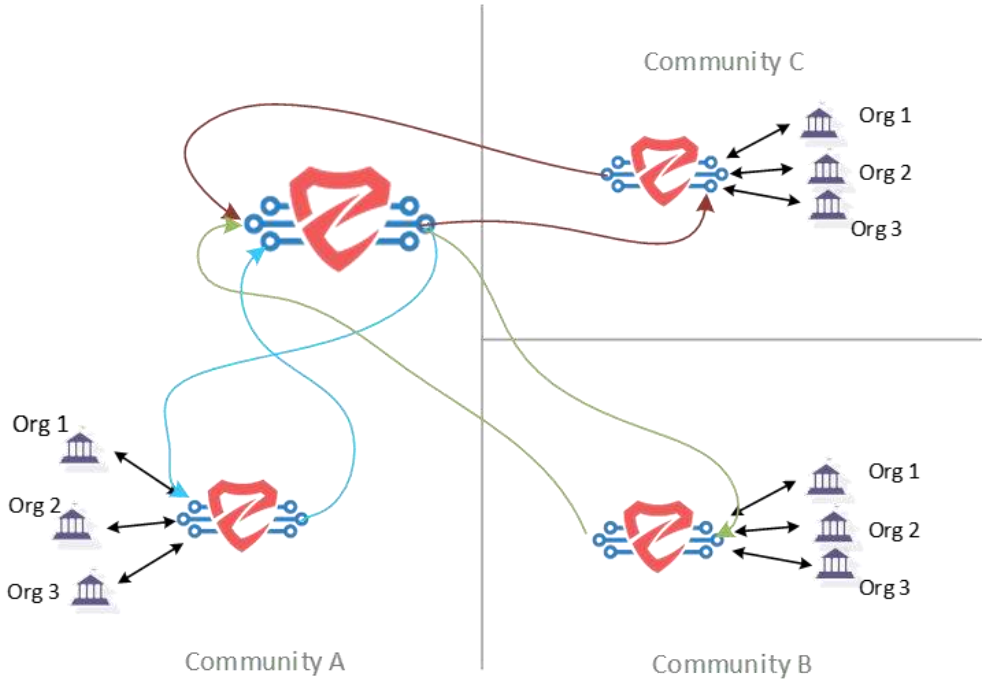

Figure 3 Centralised Architecture

3 PROTECTIVE Node

The PROTECTIVE system is designed as an information processing pipeline. Information enters the Alert Flow Processing system at the bottom left side of Figure 4 through the Alert Ingestion subsystem. Using connectors, the subsystem receives data from both internal network sources e.g. IDS, IPS, firewalls, network probes, system logs, and honeypots and also from third party sources such other security alert systems e.g. IntelMQ or n6 as well as e-mail reports etc. Threat intelligence is also received from other nodes to form a PROTECTIVE community. This community comprises all the project participants that are running a PROTECTIVE Node and other communities such as MISP. In order to connect with external communities, connectors can also be developed to connect with other communities, in this case, to connect to a MISP community. The incoming alerts are converted to the PROTECTIVE normalised security alert format, IDEA (Intrusion Detection Extensible Alert). After ingestion, IDEA alerts are passed to the Enrichment subsystem. Here, additional data is annotated to the alerts to aid with their further processing.

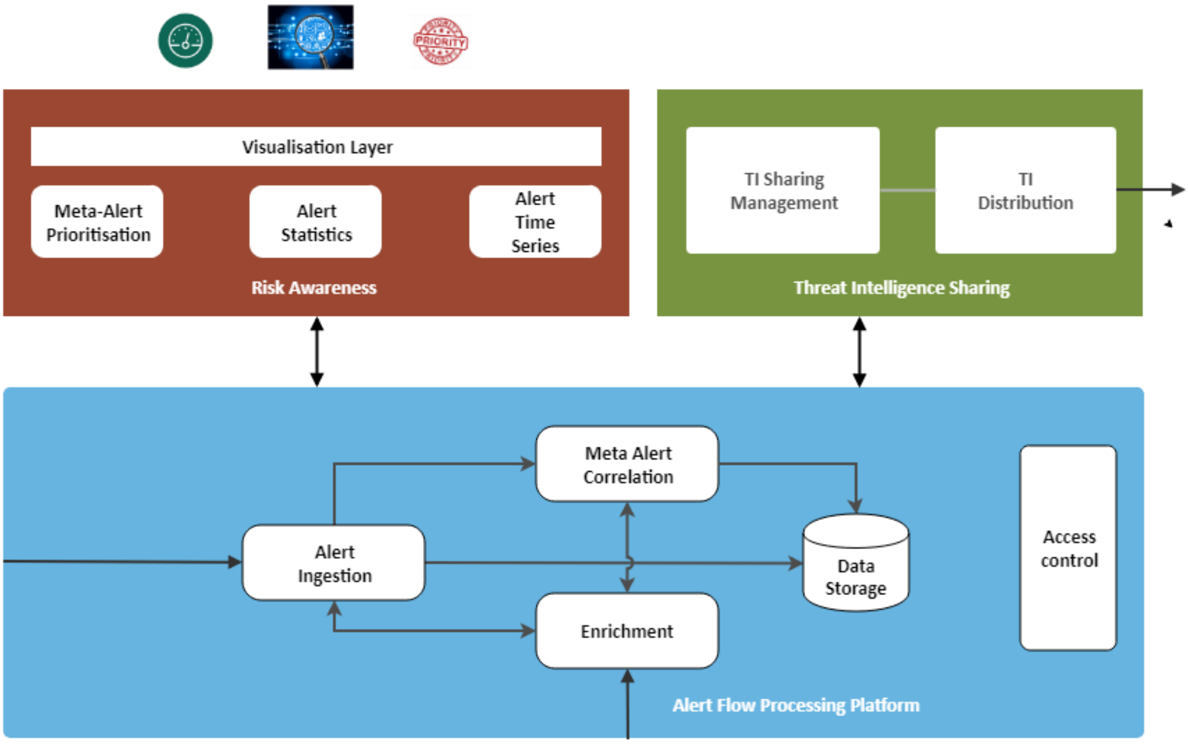

Figure 4: Software Template Architecture

After enrichment the alerts are passed to the Meta Alert Correlation for further processing. In this module IDEA alerts are aggregated into composite structures known as Meta-Alerts. Meta Alerts are grouped on the basis of pre-defined rules and typically are alerts form the same source directed towards a single target that occur within a specified time window. Alert and Meta-Alerts are stored in the Storage system, currently a PostgresSQL database.

The alerts and Meta-Alerts are used by the analysis features: Meta-Alert Prioritisation, Alert Statistics and Alert Time Series.

Security alerts are routed to the TI Sharing subsystem for distribution. This subsystem supports the administration of a number of communities with whom information is shared. Each such community has a policy-set that determines what information is shared with whom and how TI should be

processed accordingly. Information may be converted to external (e.g. STIX, MISP) formats for sharing outside the PROTECTIVE ecosystem.

In the next sections a full description of the PROTECTIVE system.

3.1 Alert Processing Pipeline

This section describes the main alert processing pipeline of the PROTECTIVE Node. It is structured as follows:

- Overview
- Ingestion
- Inspection
- Enrichment
- Correlation
- Context Awareness Enrichment
- Storage

3.1.1 Overview

The Node's main task is to process incoming alerts -- events described in the IDEA format. This is done in multiple steps, from ingestion, through enrichment and correlation, to storage. Each task is done by a separate module, the alerts go from one module to another along the pipeline.

See

Annex C: IDEA Format in PROTECTIVE for a list of PROTECTIVE's version of IDEA message fields at ingestion and after enrichment.

Figure 5 shows a schema of the default set of modules in PROTECTIVE Node.

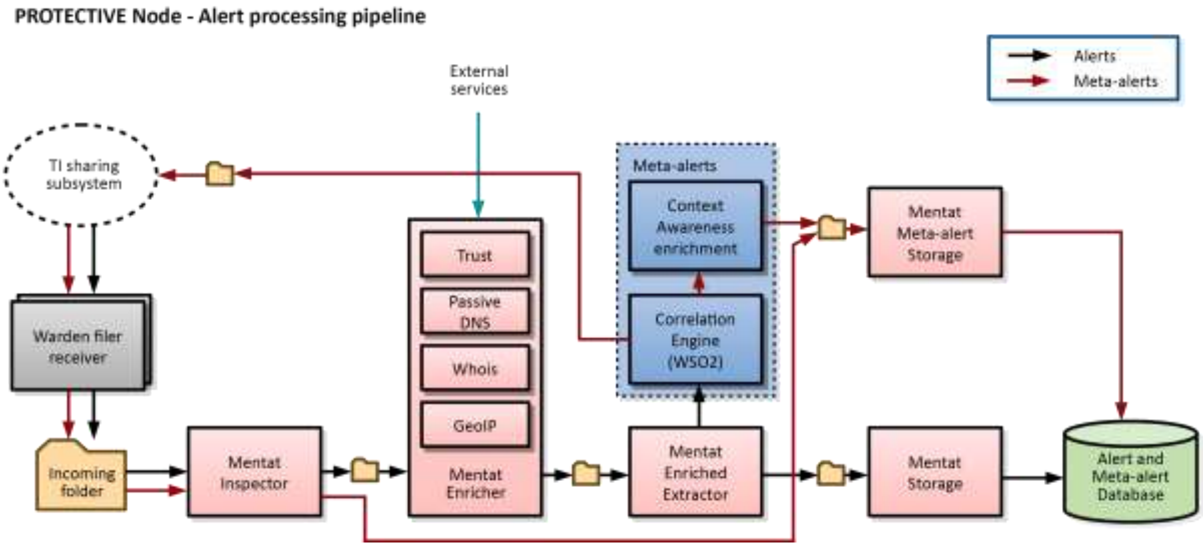

Figure 5: Alert Processing Pipeline

This modular processing system is based on Mentat software, some of the modules comes directly from Mentat, and others were added by PROTECTIVE.

The messages are passed between modules as files stored in shared directories, i.e. one module writes IDEA messages as individual files into a specific directory, other modules watch the directory and reads and removes the files.

(Note: In fact, each directory has several subdirectories, like "incoming" and "tmp", in which the files are stored. This is to prevent reading of not-fully-written files and to allow multiple readers working in the same directory; see Mentat documentation for more information.)

(Note about performance: Usually no data are actually written to hard-drive, since all disk operations are normally cached in RAM and the files are usually read and removed shortly after they are created. Therefore, this is a very simple and efficient way of passing messages between processes.)

The following subsections briefly describe individual stages of the pipeline:

Ingestion

The alerts to process by the pipeline are read from files in a specific folder. The folder is filled by one or more Warden Filer Receivers, which continuously download data from Warden server(s) (see TI Sharing section for more information).

Also, a set of connectors to ingest data from different data sources has been developed for PROTECTIVE. For a detailed list of the connectors, see Connectors.

Inspection

In general Mentat's Inspector module can be used to filter, route or tag messages according to a set of defined rules, i.e. if specific fields of the IDEA message matches defined filter expression, a new field can be added, the message can be dropped, or it can be directed to one of multiple outputs.

In PROTECTIVE default configuration, it is used to:

- Send received meta-alerts directly to storage (they are not enriched nor correlated again)
- Add some tags to alerts (event class and severity) based on a set of predefined rules

Enrichment

The Enricher module fetches additional information about IP addresses in alerts and adds it to the IDEA messages. There is a submodule for each type of information.

Currently the following enrichment submodules are part of the PROTECTIVE Node:

GeoIP - For each source IP address in an alert it resolves it ASN and country of origin (using GeoLite2 data from MaxMind) and stores them into `_CESNET.SourceResolvedASN` and `_CESNET.SourceResolvedCountry` keys.

Trust - IT adds several fields assessing the trust level of the message and reputation data associated with source IP address (es) in the Network Entity Reputation Database (NERD). See dedicated Trust Module section for more information.

PassiveDNS - For each source IP address in an alert it resolves which domain names were associated with the IP within the last week according to a passive DNS database. Two passive DNS databases are supported as source of data.

One operated by CESNET (enabled by default). For each source IP address in an alert it looks up relevant abuse contact using the configured whois service and stores it into `_CESNET.ResolvedAbuses` key.

And one operated by The Email Laundry (disabled by default, Application Progaramming Interface (API) key is needed to enable it). The results are stored under the `Enrich` key as follows:

```
"Enrich": [
  {
    "Key": <IP address>,
    "Type": ["PassiveDNS"],
    "Ref": <ID of data source>,
    "DNS": <list of domain names>
  }
]
```

Correlation

The alerts are passed into the correlation subsystem, where they can be correlated into meta-alerts. The correlation subsystem has two main components, there is an "extractor" which is implemented as a Mentat module and it is in charge of sending a copy of each enriched alert into the correlation subsystem WSO2.

See Alert Correlation section for more information about the correlation engine.

Meta-alerts resulting from the correlation process are shared to other Nodes via TI sharing subsystem and sent into Context Awareness Enrichment module.

Context Awareness Enrichment

Correlated meta-alerts are enriched by data from Context awareness component.

See Context Awareness: Ranking Implementation in PROTECTIVE section for more information.

Storage

After the alerts pass through the whole pipeline, they are stored in PostgreSQL database by the Storage module.

Meta-alerts, enriched by CA information, are stored there as well (in a separate table of the same database).

All the alerts and meta-alerts in the database are then available to GUI.

3.2 Alert Correlation

In PROTECTIVE, the correlation of alerts consists of combining alerts into meta-alerts. There are two main components in alert correlation:

Correlation Pipeline

WSO2 CEP, a third party product, is used as a pipeline that acts as a connector between the Mentat queue and the correlation engine. A custom Mentat module (Mentat-enriched-extractor), forks the enriched events to the correlation engine, WSO2 CEP, in an appropriate format.

The correlation pipeline also listens for the incoming meta-alerts generated by WSO2 CEP. When meta-alerts are received, the correlation pipeline ingests them into Mentat's queue again with a custom module (Mentat-meta-alert-storage) and Mentat stores these meta-alerts into the PostgresSQL database.

WSO2

WSO2 is a third party technology that is used to perform the correlation. It reads the input events from the correlation pipeline, applies distinct rules to them, and generates (if needed) a meta-alert, that is sent to the correlation pipeline again. WSO2 offers the user the possibility to create additional rules to the predefined ones. Figure 6 below illustrates the current set of predefined rules for PROTECTIVE.

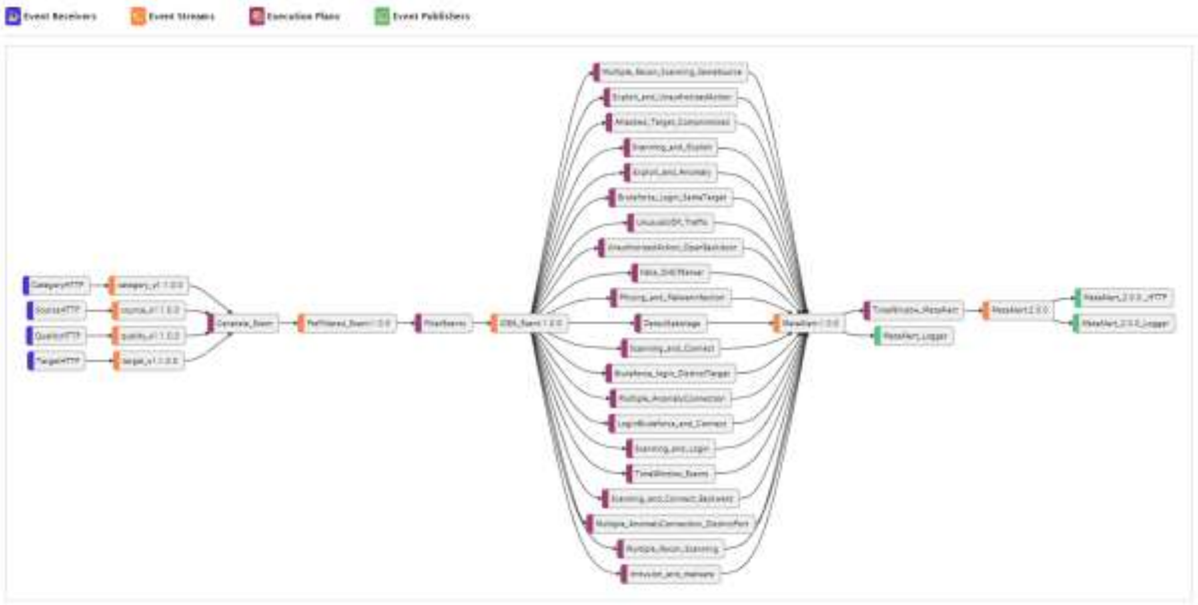

Figure 6: WSO2 predefined rules

3.3 Context Awareness

In PROTECTIVE, the Context Awareness (CA) feature is intended to enhance the performance of Meta-Alert Prioritisation by providing relevant information about the networking and computing infrastructure of the organisation i.e. the so-called constituency. Specifically, it provides two types of information:

- Measures of the relative importance of the various compute and network assets i.e. measure of asset criticality. The criticality of an asset is obtained by establishing the degree of dependency that organisation goals/objectives/missions have on a particular asset. This information helps a CSIRT assess the potential impact to the business that an attack on a particular asset may cause.
- Software vulnerability information about a compute asset (host/server)

CA provides this information to the Alert Correlation subsystem on request – potentially for every alert that is processed. CA is an optional feature i.e. the MAP function will work without CA being enabled.

The high-level design of the system is shown in Figure 7.

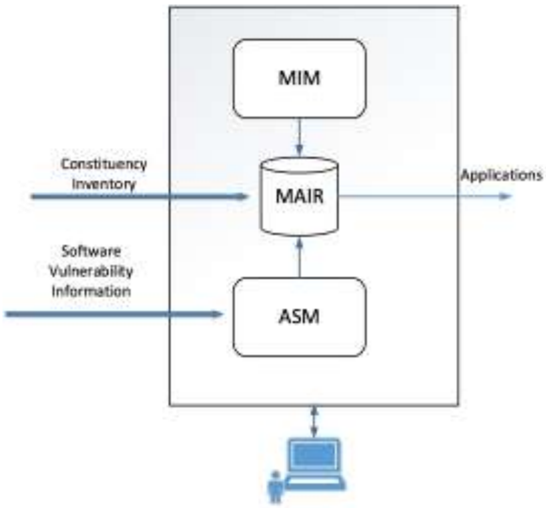

Figure 7: CA High Level Model

The CA system is described in below. It is composed of several complementary components:

- Mission Impact Management subsystem (MIM)
- Asset State Management subsystem and (ASM)
- Mission and Asset Information Repository (MAIR)

The MIM subsystem keeps track of the criticality relationships between organisation mission, or business objectives, and the network and computer assets and provides information to queries from the security meta-alert prioritisation subsystem about mission impact and asset criticality.

The ASM subsystem keeps track of vulnerability information and provides this information to the security alert handling module when queried. ASM fetches vulnerability information from the National Vulnerability Database (NVD).

Both subsystems utilise a shared Mission and Asset Information Repository (MAIR) which is a repository of the mission, services, software and hardware assets- (i.e. network devices and computers) of the organisation. This database is populated with asset information from external, organisation asset repositories through a CA defined Application Programming Interface (API). Information on missions and dependencies between missions, services and assets is also stored in the MAIR via the MIM subsystem. How to capture this relationship information is one of the major challenges for an organisation which wishes to use the CA and one approach is discussed below.

This functional design described above is implemented as shown below:

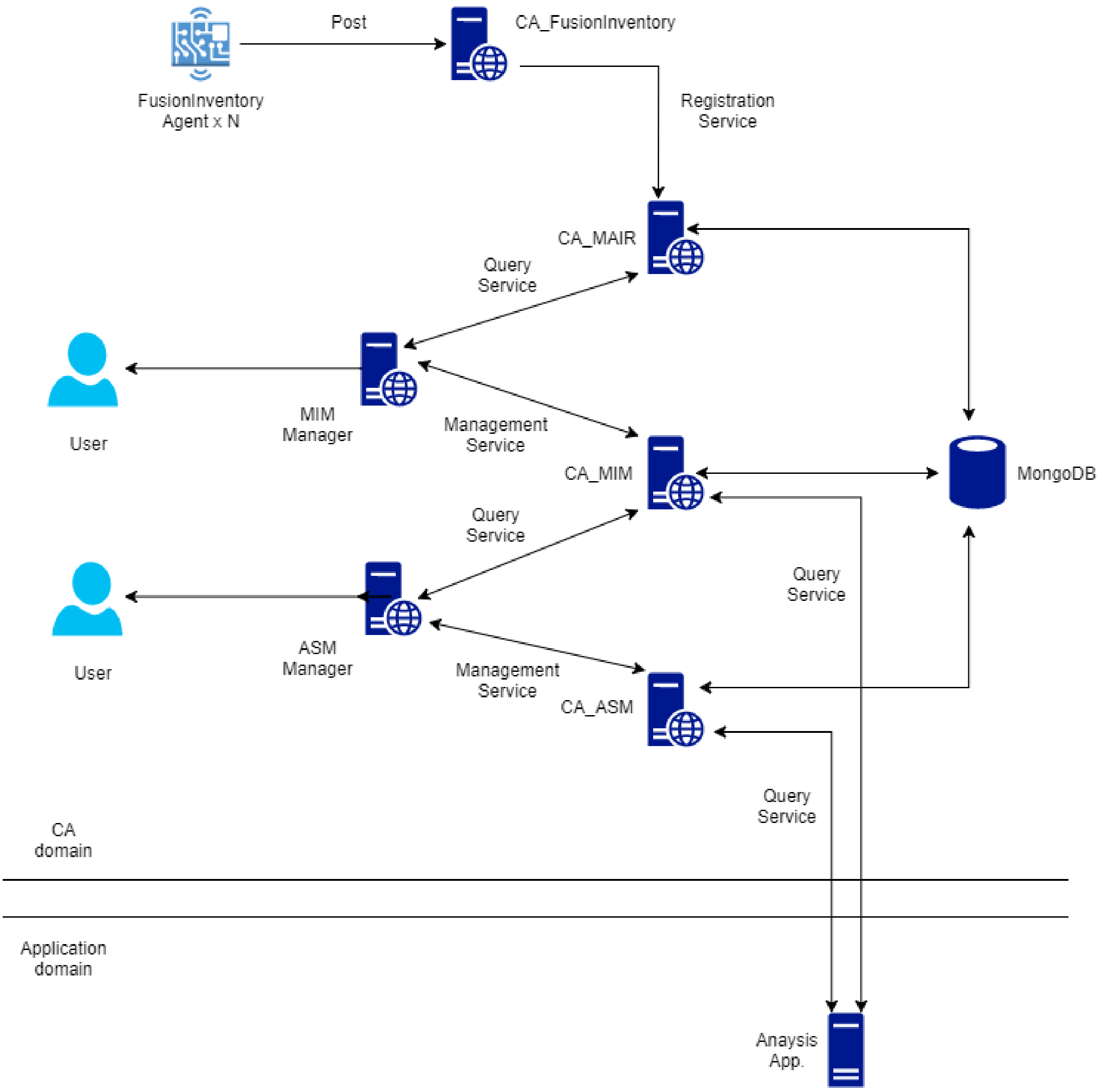

Figure 8: CA Software Architecture

CA comes with an out of the box inventory interface via FusionInventory - an open source inventory software that can be used to collect inventory information from computers though placing a FusionInventory Agent on each computer. You can find more information here http://fusioninventory.org/get/

Inventory data is uploaded via the CAFusionInventory (CAFI) container. CAFI acts as a FusionInventory server and adapts the collected inventory files to the CA-MIM format. CAFI can be used as a base for other inventory agent designs.

CAFI forwards the data to the CA-MAIR agent (encapsulates the MIM logical entity – the "MAIR" designation is historical). CA-MAIR provides a single REST interface that is used in all use-cases by CAFI, Prot-dash and the PROTECTIVE applications (described in APIs section). CA-MAIR also carries out the assets ranking. It also acquires the CVSS scoring information from CA-ASM.

3.4 Trust Module

The Trust Module is responsible for assigning to each alert a trust score that reflects how much the system believes in the accuracy and importance of the alert.

Two modules are responsible for assigning trust qualities to an alert: The trust module computes a trust score using the quality, certainty and source trustworthiness of the alert. The NERD module uses information about malicious IP addresses and a machine learning model to predict the probability of an intruder, the source of an alert, to reappear within a time period.

Figure 9: Trust module interaction illustrates how these two modules interact to enrich an alert:

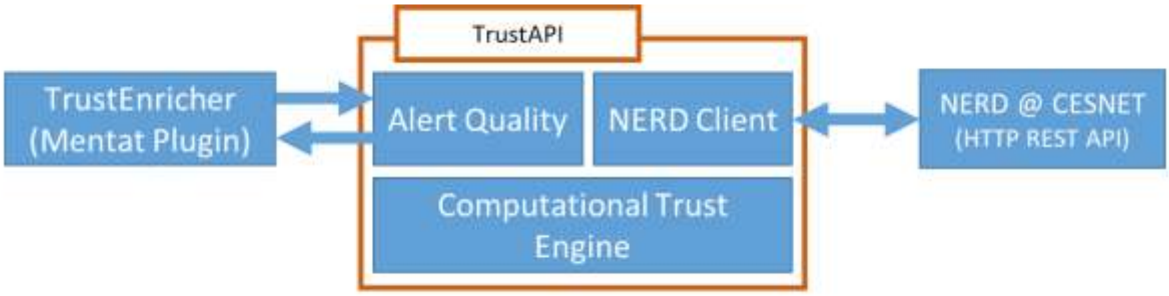

Figure 9: Trust module interaction

In the image, three components interact to process alerts. The Trust Enricher forwards alarms to the Trust Module. The Trust Module queries the NERD module using an HTTP API to discover whether the source of an alert (an IP address) is expected to reappear in the near future. With these information, the Trust Module enriches the alarm with a trust score and sends it back to the Trust Enricher, where the alarm is forwarded through the alert processing pipeline (see Alert Processing Pipeline).

NERD FMP module

NERD is a reputation database keeping various information about known malicious IP addresses. A new module for NERD has been implemented as part of Protective project. This FMP module uses information about malicious IP addresses stored in the NERD's database and a machine learning model to compute the Future Misbehaviour Probability (FMP) score. This score reflects the probability that a given IP address will perform a malicious activity in the next 24 hours.

The module does not run as part of PROTECTIVE node, it rather runs as part of the main instance of NERD operated by CESNET (which utilizes more data sources than only alerts shared within Protective). The following image illustrates the architecture and communication between the NERD client (part of the Trust Module) and the NERD server.

3.5 Meta-Alert Prioritisation

Meta Alert Prioritization (MAP) module assigns priorities to the meta-alerts selected (e.g., by date, category, etc.) from the database.

For the purpose of initial configuration, five ordered priority classes have been defined (priority one denotes the most important, priority five the least important meta-alert). The priorities are assigned to the meta-alerts in the classification process taking into account representation of meta-alerts in terms of multiple criteria.

Criteria are derived from the attributes of meta-alerts. An example of such criterion is *asset criticality* (source or target asset criticality) provided by Context Awareness module and incorporated in meta-

alert object. Some of the criteria are temporal (e.g., dependent on actual time), so the priorities are calculated on-line when the meta-alerts are being fetched from the database.

The MAP modules consist of three sub-modules:

- Criteria-mapper
- Rule-inducer
- Ranking-generator

Criteria-mapper

It serves as a proxy between meta-alerts database, ranking-generator and GUI. It provides capabilities that allow the mapping of the space of meta-alerts attributes into the space of criteria used in the decision-making process and classification. It also allows selecting, from the database, sets of meta-alerts for prioritisation. More information is available in criteria-mapper documentation.

Rule-inducer

Allows to configure meta-data, upload data (meta-alerts), induce or upload decision rules (previously induced or provided by a domain expert), and validate and inspect them. Meta-data provide formal definitions of criteria that are used for classification, as well as decision classes. Set of meta-data defines types of criteria (e.g., gain, cost) and their domains. More information is available in rule-inducer documentation.

Ranking-generator

Uses meta-data and decision rules to prioritise meta-alerts described by defined criteria. Meta-alerts, which are prioritised, are previously processed by criteria-mapper. More information is available in ranking-generator documentation.

The following items are needed to run the MAP module, assuming that the meta-alert structure is fixed:

- Definitions of criteria – in the form of meta-data used by rule-inducer and ranking-generator.
- Mapping of values of meta-alerts' attributes into values of criteria – mapping template used by criteria-mapper. The mapping must be in sync with meta-data. The change of meta-data must be followed by an appropriate change in the mapping template.
- Prioritisation model in the form of decision rules (note that in DRSA, single rule suggests assignment of a meta-alert to a union of ordered decision classes, not to a single decision class). The criteria and decision classes used in rules must correspond to the ones defined in meta-data.

Also, within the criteria-mapper module, JSON to HTML conversion templates are provided to allow simple, tabular visualization of the results and to simplify collecting of preference information. Note, that changes of meta-data will require changes of object definitions in the visualization templates.

In order to speed up deployment, a set of criteria, mapping templates and a basic set of rules have been predefined and delivered with docker images.

All the configurations above and templates are stored in the PostgreSQL database and can be changed using APIs provided by the submodules or directly in the database.

3.6 Analytics

3.6.1 System and Sensor Data Statistics

In order to aid the operator in their day-to-day tasks and activities, Prot-Dash assures the implementation of a general overview of the current status of the PROTECTIVE system instance they are operating as well as the most recent events.

This view, is based on a multi-layered approach. The first layer is represented by a high-level dashboard that includes the most important parameters, and that allows the operator to drill down into relevant activities. The second layer provides additional details about whatever component the operator chooses to investigate. Visualisations shown on the dashboard are completely configurable but default to displaying the most recent alert information.

Figure 10: PROT-Dash home

In Figure 10: PROT-Dash home, the home of PROT-Dash, shows a default overview, that includes the following widgets:

Alerts per source

This statistic provides information which source exports the majority of alerts as well as it may reveal a failure of a source if the source disappears from this statistic suddenly. This time series plot shows a stacked bar chart. The y-axis indicates the number of alerts that have been recorded. The x-axis shows the time at which the alerts were detected. This time granularity is adjustable. Each bin represents a time period for which alert data was detected. We can see that each bar is multi-coloured and each colour represents a source from which the alert is coming.

Alerts per partner

This is similar to the Alerts per source graph but in this case, the different colours in the graph represent the partner from which alert originated.

Alerts per category

This graph, as the name suggests, is illustrating the category of attack using the colours in each of the bars. It reveals which attack categories are popular and in case there is an unexpected spike it may

indicate an anomaly.

Source status

Finally, the table gives more specific information about the number of alerts being ingested at the time recording. The dashboard overall gives a good overview of the threat status of the network and which partners are more at risk.

Custom graphs can be added from Statistics view. In this view the operators can generate their own custom graphs and save it or load the stored dashboard they created previously. Here, the operator will be able to generate custom statistics. The first action operators have to do is to select a View Provider. The two principal View Providers are Plotly timeline and Chart JS. Additional View Providers are listed in the dropdown menu but will not be described here.

Plotly Timeline

Plotly Timeline is the that allows the user to generate custom time series' graphs, compute the trend algorithm of a set of events and show alerts if the computed value differs a certain quantity from the real number of received alerts.

Once "Plotly Timeline" View Provider is selected and "Add widget" is clicked, operators click edit the widget and in the General tab select the Data Provider "Neon Time Series Framework". In the Data Provider tab the user is presented with a form which is filled in to generate custom stacked time series graph as shown in Figure 11: Timeseries form.

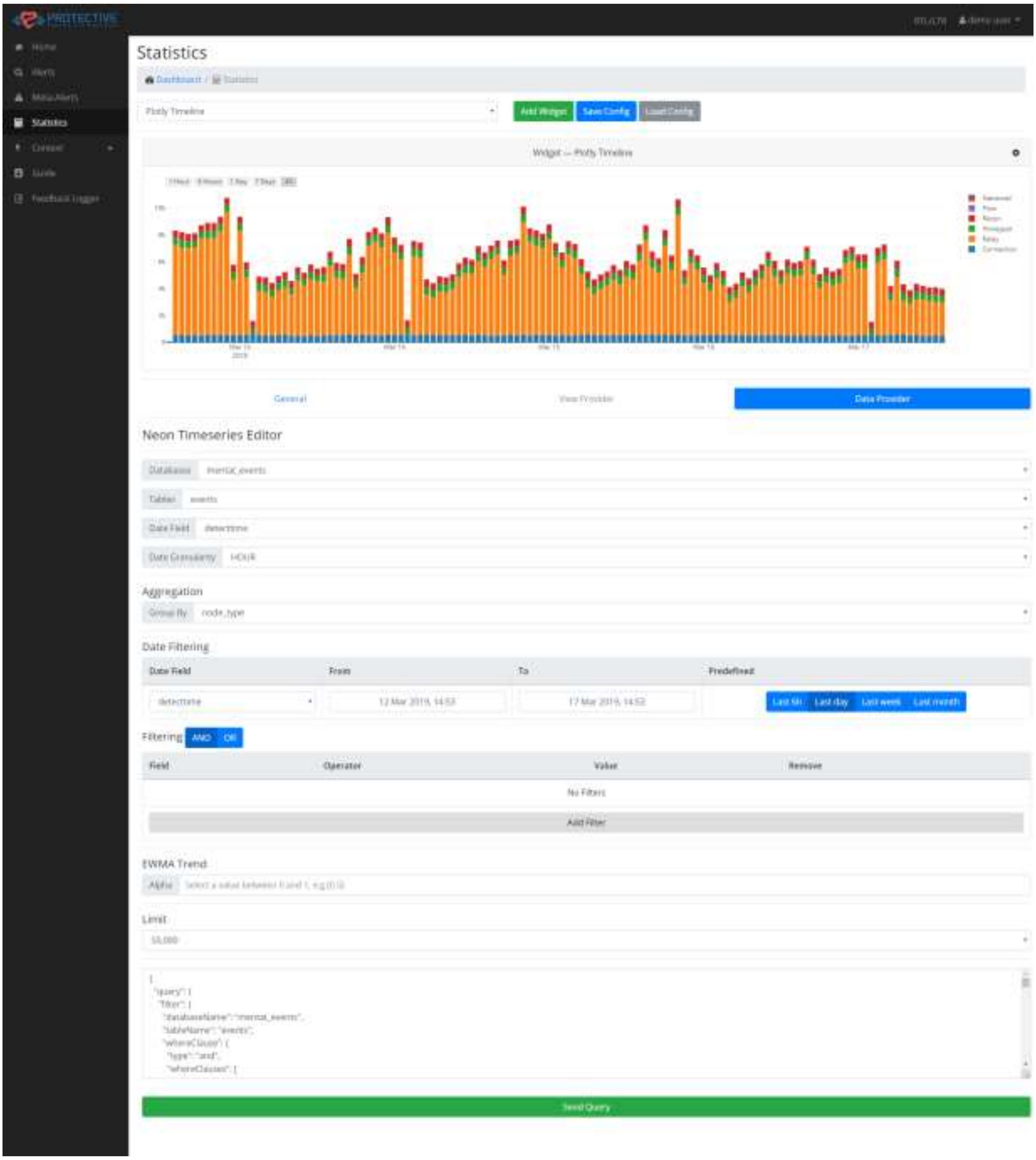

Figure 11: Timeseries form

When all the fields are filled and "Send query" is clicked, the graph will appear.

This Data Provider allows the user to specify the decay factor, alpha to compute the Exponential Weighted Moving Average (EWMA) trend for the resulting graph. An alert system has been implemented for this computation. When the current value is 3 times larger than the predicted value and current value is at least 30, an alert will be generated. These generated alerts can be found in a dropdown that appears just above the graph, as shown in Figure 12: Trend and alerts.

Figure 12: Trend and alerts

Chart JS (Multiple)

Chart JS views are useful for making any type of graph that does not require time series such as a bar chart showing the count of Categories, or a Brief (Pie Chart) showing all the events grouped by node software.

Once "Chart JS" View Provider is selected and "Add widget" is clicked, operators click edit the widget and in the General tab select the Data Provider "Neon Framework". In the Data Provider tab the user is presented with a form which is filled in to generate custom graphs or briefs. This View Provider allows the operator to change the type of graph presented in an easy way. On the widget edition in the View Provider tab the operator can switch the type to bar chart, line chart, pie chart, etc and also he can hide some of the results for better detail. In Figure 13: Brief by Category, a brief of the alerts by category is shown.

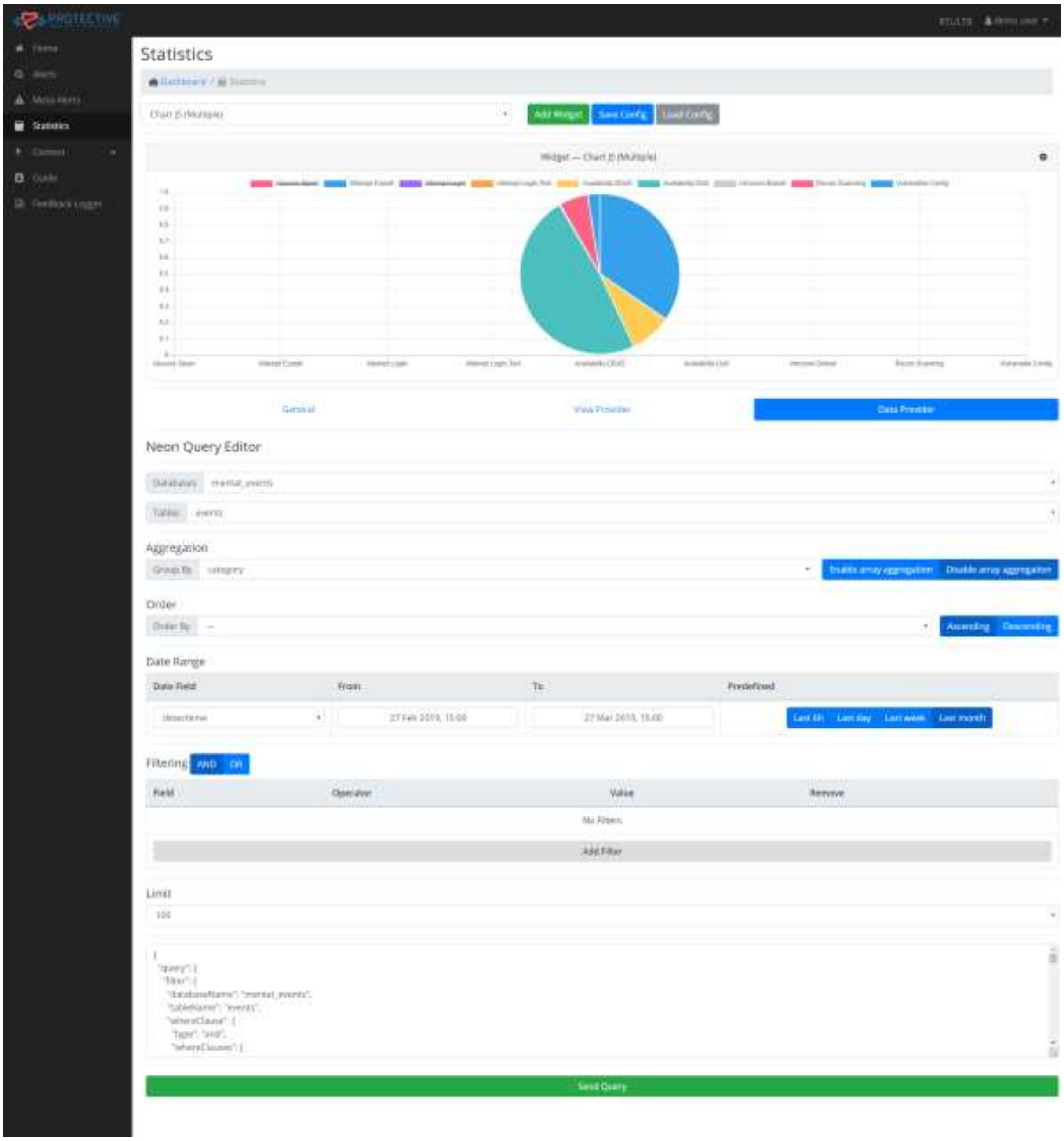

Figure 13: Brief by Category

3.6.2 Time Series and Trend Monitoring

The purpose of the time series is to provide an operator with a visual overview of the development of trends over time and to identify areas requiring particular attention, such as a spike or unexpected decrease of monitored characteristics.

This section is strongly interrelated with the above in terms of what data is necessary to be collected but instead of the default dashboard time series plots that have a pre-configured time period and other pre-set query parameters, here we have full control over the parameters we include in our query. We can specify which time field we use for the time period, we can specify the granularity of the data we wish to have returned, we can group the data by a specified field and filters can be added using any field available in the database.

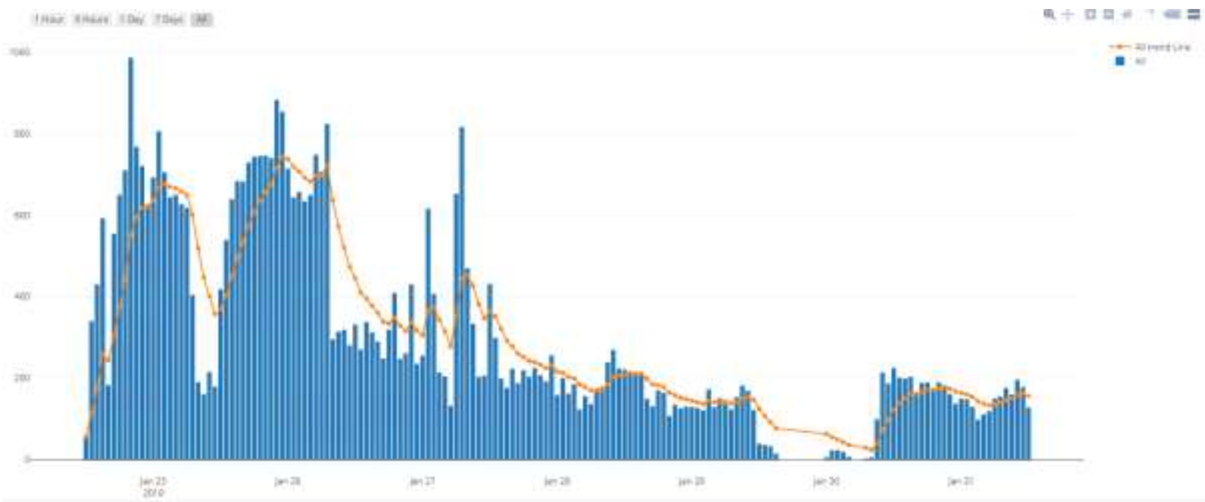

Figure 14: Trend monitoring

3.6.3 Prediction of Future Events

PROTECTIVE provides two features to help the CSIRT predict or anticipate events that are more likely to occur in the near future.

Future Misbehaviour Probability

NERD is a reputation database keeping various information about known malicious IP addresses. A new module for NERD has been implemented as part of Protective project. This FMP module uses information about malicious IP addresses stored in the NERD's database and a machine learning model to compute the Future Misbehaviour Probability (FMP) score. This score reflects the probability that a given IP address will perform a malicious activity in the next 24 hours.

The module does not run as part of PROTECTIVE node, it rather runs as part of the main instance of NERD operated by CESNET (which utilizes more data sources than only alerts shared within Protective). Figure 15: NERD Client-Server Communication illustrates the architecture and communication between the NERD client (part of the Trust Module) and the NERD server.

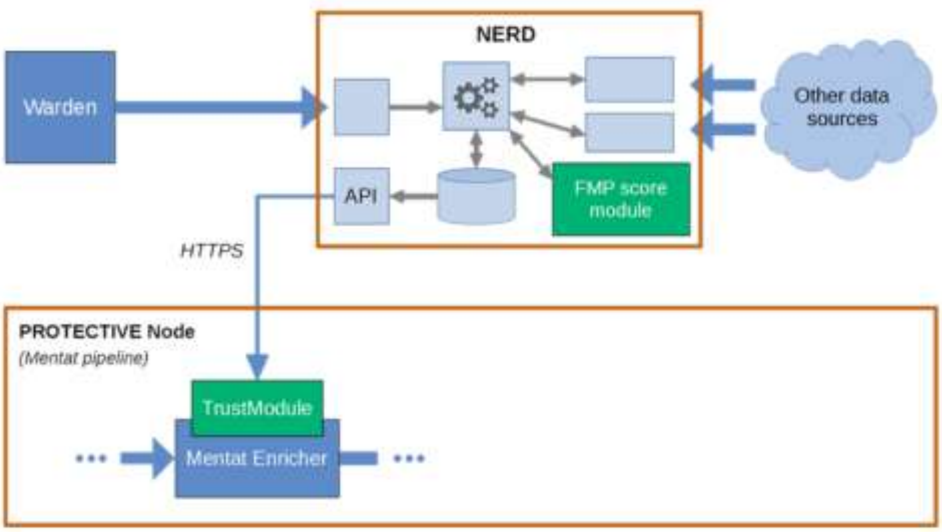

Figure 15: NERD Client-Server Communication

The NERD client fetches FMP score for each source IP address in an alert and stores it as the EntityReputation key.

Alert Prediction Using Deep Learning

The project is currently developing a function, based on Recurrent Neural Networks (RNN) to predict the probability of certain sequence of alerts occurring within the immediate future. The deep learning model uses the proven long-term sequence learning capability of Gated Recurrent Units (GRU) to learn the behavior of attacking sources, and thereafter predict future alerts originating from such sources. The system is different from existing approaches in two ways. Firstly, this is the first attempt in which entire alerts (including fields such as DetectTime, FlowCount, Port, Protocol, etc.) are predicted rather than the probability of an alert occurring in a future time frame. Secondly, while most prediction systems are designed to predict the probability of attack against a given target, the alert prediction system in this case is designed to predict alerts corresponding to attacking sources. This model may be used in addition to NERD, to provide a more holistic approach for attack prediction. First, NERD can be used to predict how likely it is for a source to perform an attack in a near future. If the probability is high and there are enough previous alerts, the deep learning method can then be used to predict the expected parameters of such an attack.

The model is trained by presenting alert data in the form of history and future windows. For instance, a typical training sample may be comprised of 20 past alerts and 5 future alerts originating from a particular source. The trained model is then used to predict future alerts for different sources given only the history window of alerts. Initial results show that the alert prediction model demonstrates better capability to predict alerts from very active attacking sources. For instance, the model provides a prediction accuracy of 84% for sources which attack more than 100,000 times in a month. Similarly, for sources which report 10,000 or more alerts in a month, future alerts can be predicted with an accuracy of 80%. For sources which are not so active in producing alerts, the deep learning model is capable of predicting future alerts with an accuracy of approximately 60% for sources which report more than 40 alerts per month. Efforts are ongoing to improve the prediction accuracy, and to extend the prediction capability for less-frequent sources i.e. for sources which report less than 40 alerts in a month.

3.7 Threat Intelligence Sharing

3.7.1 TI Sharing in PROTECTIVE

PROTECTIVE provides two modes of threat intelligence sharing:

- Centralised Sharing
- P2P Sharing

An overview of these is given in section 2 of this document.

The PROTECTIVE component is responsible for sharing and storing threat intelligence and for managing distribution to partners. It manages data exchange according to information sharing policies and agreed protocols and data formats. Through the Threat Intelligence Sharing, the PROTECTIVE framework is able to receive and/or send alerts in IDEA Format, from/to other peers or from any of the installed connectors at its network.

The PROTECTIVE node includes a third party product, the Warden System, that provides the main underlying functionality of the TI sharing Management and TI Distribution components which are working with certificates to establish a secure communication between the PROTECTIVE nodes. See more details about Warden System in Annex A: Warden Overview.

Each entity/network that wishes to feed data into the Warden system will have a so called sending client. Each entity/network that wishes to receive data from the Warden system will have a so called receiving client. The Warden server (the centre) ensures the data reception and storage as well as the interface for the access to data stored. Data which the clients send into the centre are referred to as events (alerts). Events are sent by the sending clients after authentication; the access to the centre is also authenticated. X.509 is used for the authentication. See more details in Security.

3.7.2 IDEA Format

This is the data-format selected for PROTECTIVE's TI Sharing. IDEA format is fully compatible with the core components of PROTECTIVE, Warden and Mentat and it is developed and maintained by CESNET.

IDEA stands for Intrusion Detection Extensible Alert. Even though there exists a variety of models for communication between honeypots, agents, detection probes, none of them is really used because of various limitations for general usage. The IDEA is an attempt to define nowadays requirements and propose foundations for viable solution for security event model, taking into consideration existing formats, their benefits and drawbacks. This format, wants to hit some middle ground between complexity of IDMEF and free spirit and structure (or lack thereof) of AbuseHelper, learn from pitfalls of existing projects and based on experience as members of CSIRT team, propose solutions to some of them on the way, taking into consideration recent evolution and requirements in the field.

To get more information about this format, the IDEA schema and definition you can visit the CESNET’s official IDEA webpage also, the PROTECITVE’s version of IDEA format can be found in

Annex C: IDEA Format in PROTECTIVE.

3.7.3 TI Sharing Management

In order to establish a communication between two nodes or between a node and a connector, Warden has a tool called Warden Filer. Warden Filer is a python daemon for easy handling of IDEA events transfer between plain local files and Warden Server. The tool can be instructed to run as one of two daemons - receiver and sender.

Warden Filer Sender or sending client: Warden Filer polls directory and sends all new files out to Warden Server.

Warden Filer Receiver or receiving client: Warden Filer polls Warden Server and saves incoming events as plain files in directory.

In both modes, the *Uniform Resource Locator* (URL) of Warden Server and the directory are specified through a configuration file.

3.7.4 TI Sharing Distribution

As described in the previous section, the minimum configuration will be a configured Warden Filer Receiver in order to receive events from the Central Warden Server and a configured Warden Filer Sender to send events to the Central Warden Server. Then, for some connectors you may need to add more Warden Filer Senders.

3.7.5 Information Sharing Compliance

The Information Sharing Compliance (ISC) module examines outgoing threat intelligence in order to detect and act on content that violates policy. The ISC is intended to provide a safeguard against accidental data leaks that may exist in shared threat intelligence.

Such potential leaks might include accidental sharing of GDPR-protected data, of corporate confidential information, of private key material, or of information such as private email addresses or internal IPs. The ISC is intended as an independent module of PROTECTIVE, so that it can act as a final check on data being shared through the system. The ISC runs in a separate container and all alerts and meta-alerts are passed through the ISC before being shared.

It operates by applying a configurable rule-set to each outgoing alert and meta-alert, and performs actions to ensure compliance. The ISC module is able to perform one of four actions:

- Keep the alert unmodified, where no violations were detected.
- Drop the alert entirely, so that it is not sent out at all.
- Erase the problematic content, replacing it with a placeholder.

Report the rule violation in the log file but otherwise send the alert unmodified.

3.8 Security

3.8.1 Node Connectivity

In order to be able to connect a PROTECTIVE node with other nodes, with Central Warden or a with connector, an X509 client certificate and private key needs to be generated for each node/connector and stored in the Warden Filer Receiver’s certs folder of the protective node installation directory.

Server certificate of the corresponding server needs to be stored there as well. This certificate does not change, it is generated during server installation and is used for all clients. It is used by clients to

validate authenticity of the server. (It can be a self-signed certificate, in such case it has to be distributed via a secure channel to all clients, which then set it as trusted. It can also be a "normal" certificate signed by some commonly-trusted CA, like Let's Encrypt, then the client can rely on its OS capabilities to check the server certificate.)

This allows a secure HTTPS communication between the local Warden Filer Receiver and each of the clients.

The procedure in order to get the certificates involves sending a PGP signed email to the Peer node's management team requesting access to it and with some data of the node. The details of the email can be found in the installation guides under Installation section. Then, if the Peer node's management team accepts your request, he will send the needed certificates and the URL of the node.

As a note, the signature of the email, ensures that the identity of the requestor, as well as integrity of the request, can be validated. Then it is up to the server/node operator to decide if that person is allowed to connect the specified node or connector. This way, the server/node operator has the responsibility to detect potential malicious user requests and deny them when needed.

<u>**Secure communication between nodes**</u>

In order to generate and manage certificates for secure communications, PROTECTIVE relies on a main component:

Warden Registration Authority (RA) for Warden

Warden RA is a certificate registration authority for Warden_ server. It is meant to support the client registration process and simplification of the credential transport.

As Warden clients are authenticated by X509 certificate, the usual certificate generation process can be used - a local private key and a certificate request gets generated, the request is submitted to the registration authority, and after review, a signed certificate is issued and delivered back.

A CA (certification authority) certificate of Warden RA is used to sign the certificates. This CA certificate is set as trusted in Warden Server, so any client certificate signed by it is trusted.

Certificates generation

As you will find in the installation guides, this process has been automatized by the execution of a simple script registerClient.sh which automatically generates the certificates, but for a better understanding, in the next lines we will detail the process of the certificates generation.

When connecting two nodes, either peer to peer or node to central, the process starts with a node (client) that wants to connect to another node (server). The client's administrator has to send a PGP signed email to the server management team with some information like the client's hostname and the name and email of the responsible person.

After successful validation of the client parameters, the server registration officer registers the client in Warden Server, Warden RA and enables new certificate generation by running (on the server side):

```
warden_server.py register --name org.example.client --hostname X.X.X.X --
requestor client@mail.com --read --write --notest

warden_ra.py register --name org.example.client --admins client@mail.com

warden_ra.py applicant --name org.example.client
```

The last command returns a one-time token which allows the client to get its certificate signed by Warden RA. This token is then sent (preferably through a secured channel, such as PGP encrypted and signed email) to the new client administrator along with other setup information and certificate of the Warden Server.

The client administrator runs the application script with the received one-time token:

```
warden_apply.sh org.example.warden.client TOKEN
```

The script automatically creates a new pair of keys, X509 CSR (certificate request) and makes a call to the Warden RA web service. Warden RA checks whether information in the CSR and the token matches those in its database and if it is correct, signs the CSR, thus creating a valid PEM certificate which is sent back to the client. The client then uses this certificate to authenticate against the Warden server.

When installing new connectors to the node, the process is exactly the same, but in this case the client is not a node, it is the connector itself. In the next section we will provide more detail about this process.

Finally, once the certificate is generated, all the files (client key and cert and server cert) need to be stored in the certificates folder of the receiver of the node or in the certs folder of the connector. Both are folders of the protective-node installation.

3.8.2 TI Sharing Ports Diagram

In the next Figure, you can find the ports diagram that summarizes all the ports' communication on the Node. Each box is labelled with the docker container name.

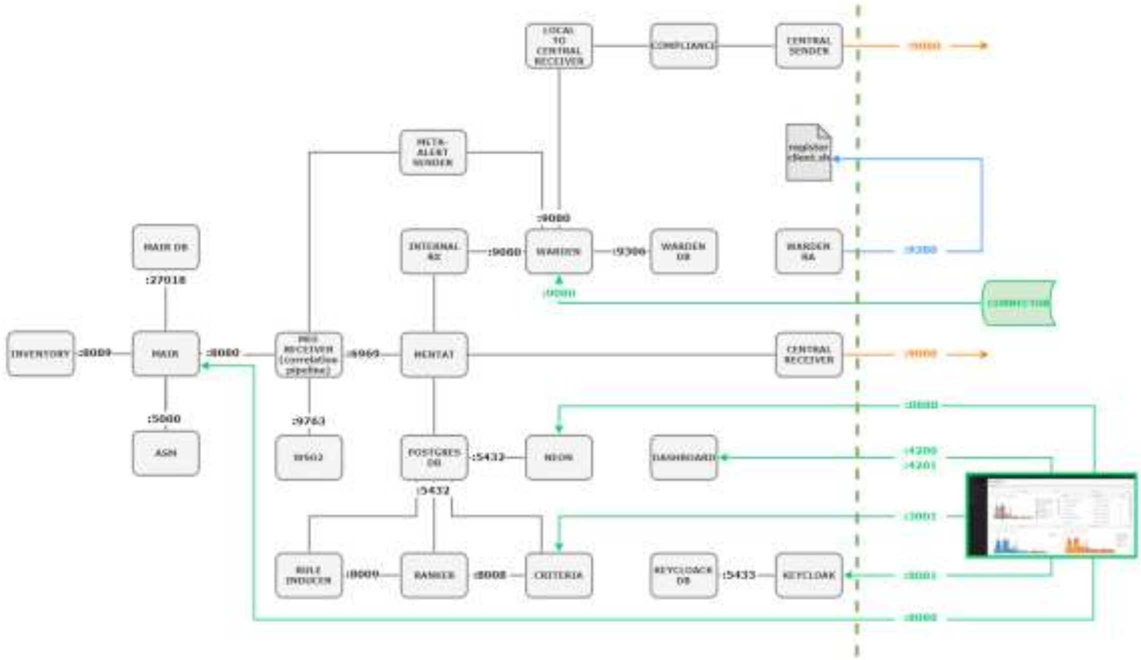

Figure 16: Ports Diagram

In the diagram we have 2 levels of connections:

- **Internal to the organisation:** Which is represented by all the ports in black. These are all the connections between the docker containers that are done internally in the organisation.
- **Connections to outside:** Which is represented by the orange (outgoing) and blue (incoming) ports. These are:
  - **The connection to send events to Central Warden:** 9080. This connection is outgoing, the node sends a POST request to store the events in the remote Central Warden.
  - **The connection to receive events from Central Warden:** 9080. This connection is outgoing, the node sends a GET request to receive the events from the remote Central Warden.
  - **The connection to generate the certificates (Warden RA):** 9280. This connection is incoming, the node sends a request to itself (using its own DNS or IP) to register the client and generate the certs.
  - **Optional connections to outside:** Which is represented by the green ports. In case you want to access your dashboard or install a connector outside your company's network.
    - **The connections for Dashboard:** 3001, 4200, 4201, 8080, 8081, 8888. This connection is incoming, the browser (located outside the node) sends requests to the different components of the dashboard to make it work.
    - **The connection of the connector to our node:** 9080. This connection is incoming, the node gets a POST request to store the events generated by the connector.

3.8.3 User single-sign on (SSO)

OIDC 1.0 is a simple identity layer on top of the OAuth 2.0 Authorisation Framework (OAuth). Four types of authorization grant are described in OAuth (see RRC 6749). For PROTECTIVE, two of these

grants are proposed: Authorisation Code (AC) for single sign on and Client Credential (CC) for microservice AA.

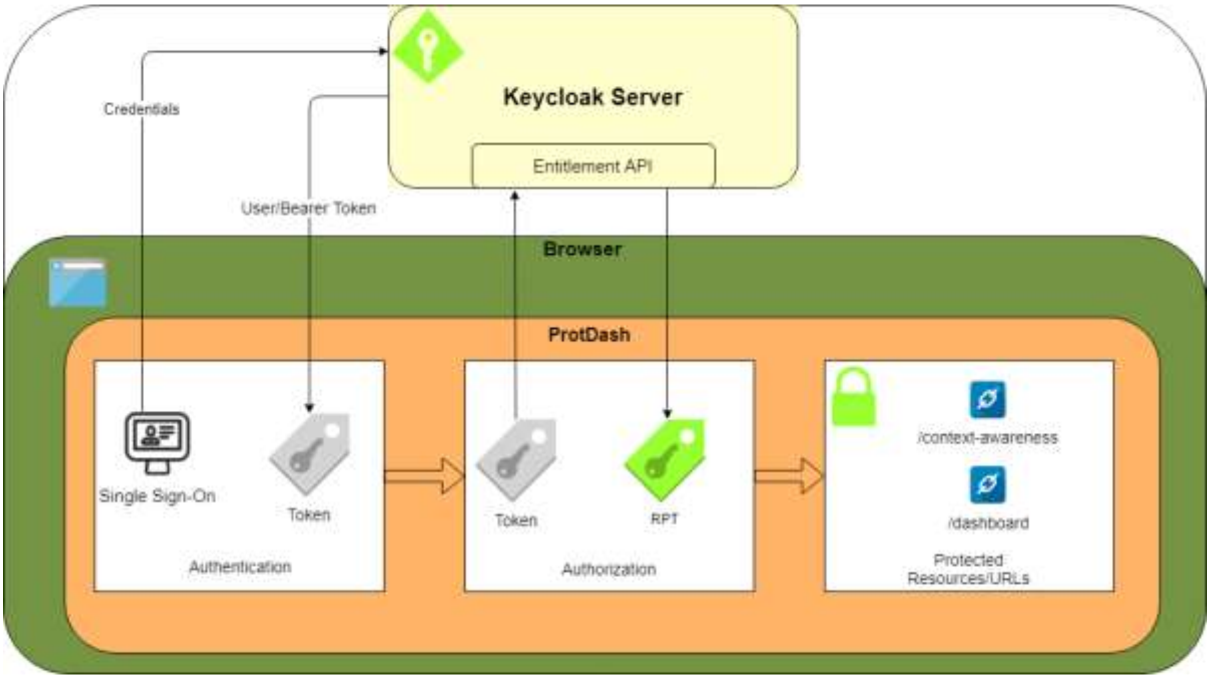

Figure 17: Keycloak integration with PROTETIVE

Keycloak (KC), an open source software product, that provides support for OIDC and OAuth, has been integrated with PROTECTIVE. The figure shows how PROTECTIVE uses KC for SSO and AA.

The following subchapters describe how PROTECTIVE can be implemented with four role types for an NREN type organisation and using the Context Awareness (CA) application as an example.

User Roles

In order to illustrate the use of Keycloak for access control, the following example is given. Four user roles (see Table below) have been defined based on PROTECTIVE being used by NREN organisations. Each role has a name, a list of Security Operations Center (SoC) activities carried out by the user on a PROTECTIVE node and TI sharing responsibilities that were used when defining the roles and operations. These roles are based on a state of the art study. Note, however, a PROTECTIVE administration can create whatever roles required.

Name	SoC Operations	TI Sharing
Manager	• grants access to Protective node to other members • configures Protective node vital parameters • full access to other Protective node features • checks who has access to data and how data are processed • provides overall direction for the SOC and input to the overall security strategy	• manages sharing policy • deals with issues raised by NRENs • develop and maintain security policies, procedures • provides best practice sharing policy
Senior Analyst	• possesses in-depth knowledge on network, endpoint, threat intelligence, forensics and malware reverse engineering, as well as the functioning of specific applications or underlying IT infrastructure • closely involved in developing, tuning and implementing threat detection analytics • performs deep-dive incident analysis by correlating data from various sources • advises on remediation • reviews trouble tickets generated by Level 1 Analyst(s) • leverages emerging threat intelligence (IOCs, updated rules, etc.) to identify affected systems and the scope of the attack	• ensure timely and accurate incident sharing and documentation • manages connecting of alert sources • manages NREN connection and TI routing •
Junior Analyst	• first point of contact for events in a SOC • executes daily operations and procedures as a matter of daily responsibility • responsible for monitoring the SOC main channel for security events and closing or escalating those events as necessary • triage the security alerts • creates new trouble tickets for alerts that signal an incident and require response review • monitors health of security sensors and endpoints	• N/A
Constituency Member	• receives alerts/meta-alerts/reports that are relevant for her/his organization • may access Protective node and query subset of data relevant for her/his organization	N/A

Figure 18 PROTECTIVE User Role Example

The Keycloak (KC) product has its own definition of roles:

- **Realm role:** KC defines a realm to be a domain for a set of clients, resources etc. A PROTECTIVE node is a relam. A realm role is accessible to all clients.
- **Client role:** A role defined for a specific client and usable only by that client. Clients are entities that can request authentication of a user. Clients come in two forms: an application that wants

to participate in SSO and are provided security by Keycloak; an application that is requesting an access token so that it can invoke other services on behalf of the authenticated user.

- **Composite_role**: which is an aggregation of a number of client or realm roles. Any role can be made a composite role by associating other, existing, roles with the role. One very common use for the composite role concept is to assign a set of roles to a KC User. Thus a user role is realised by mapping a KC composite role to the user.

Each KC role will have associated KC Client/realm roles which authorize access to different functions in PROTECTIVE. For details on how these are implemented in KC and how to implement your own, we have a guide for basic operations but ultimately, to take advantage of the full functionality of KC you should consult the documentation. More details in the guide describe how these KC Client/realm roles have been defined for CA. There are also details there on what user-role mappings have been carried out and how to implement your own mappings.

3.8.4 External APIs

3.8.4.1 Access to NERD

In order to get the Trust Module working with its interaction with NERD, an API token is needed.

To request the API token, an email to nerd@cesnet.cz requesting the token for your PROTECTIVE Node has to be sent.

Once the token is received, it needs to be placed in Mentat's enricher module configuration file and Mentat service has to be restarted to load the new changes.

3.8.4.2 Access to PassiveDNS

The passive DNS configuration is pretty similar to NERD, an API token is needed to enable it on the system.

In this case, an API key needs to be generated by EML and then requests can be made to Passive DNS.

Once the token is received, as in the previous section, it needs to be placed into Mentat's enricher module configuration file and Mentat service has to be restarted to load the new changes.

3.8.4.3 Context Awareness

As described in section 3.3, Context Awareness, has an API that allows the system to fetch a wide variety of data related to contextualization, such as the criticalities or the Common Vulnerability Score System (CVSS). The API calls and their description is available in the PROTECTIVE's documentation:

- **CA MAIR**: https://protective-h2020-eu.gitlab.io/enrichment/context-awareness/ca-mair/ca-mair-api
- **ASM:** https://gitlab.com/protective-h2020-eu/enrichment/context-awareness/ca-asm/blob/master/api-description-v1.03.pdf

3.8.4.4 Meta Alert Prioritisation

The components of this module, also have a set of APIs as described below:

- **Rule inducer:** Rule Inducer (rule-inducer) is a part of Meta-Alert Prioritisation (MAP) module. It is a REST service providing endpoints allowing to induce, and test decision rule models - Multi-Criteria Decision Analysis (MCDA) models. These models may be used by Ranking Generator (and Criteria Mapper) to prioritise meta-alerts. Other endpoints provided by Rule Inducer allow to store and retrieve meta-data, data, decision rules, as well as to manage

models, and users. In the next link, the list of the REST endpoints is provided: https://gitlab.com/protective-h2020-eu/meta-alert-prioritisation/rule-inducer/blob/integration-test/README.md#rule-inducer

- **Ranking generator:** This REST service is providing endpoints allowing to apply rule models (MCDA models) to prioritisation (ranking). More precisely, the following REST endpoints are provided: https://gitlab.com/protective-h2020-eu/meta-alert-prioritisation/ranking-generator/blob/1.7.0/README.md#ranking-generator

- **Criteria mapper:** Criteria-mapper modules provides following API endpoints (Note: ranking-generator module must be running and available for criteria-mapper to use most of endpoints (/criteria is only exepcion) and also database of meta-alerts must be accessible for the module): https://gitlab.com/protective-h2020-eu/meta-alert-prioritisation/criteria-mapper/tree/master-postgresql#api-

3.9 User Interface

In an attempt to give a somewhat detailed overview of the Prot-Dash visualisation web user interface for the Protective project, this chapter will detail the design and implementation choices that were made as well as some of the details of the current implementation.

3.9.1 Design choices

Single Page Application

During the design stage, it was decided that the responsiveness of the web app was extremely important to the user experience. A SPA is a web application that fits into a single page. Dynamic actions can be carried out on the page without adding long loading times by having to refresh the entire page. For example, if a table on the page needs to get fresh results from the database, it queries the database in the background and triggers the reload of the table content but not the entire page. This makes for a smoother experience for the user when clicking around different parts of the web application.

SPAs are common and used by Gmail (Google, 2018a), Google Maps (Google, 2018b), Facebook (Facebook, 2012) and GitHub (GitHub, 2018) to name a few. Modern Javascript frameworks that harness the power of AJAX make this possible. Although there are several SPA frameworks to choose from, ultimately, we decided on the use of Angular (Angular, 2018) for several reasons. Angular uses change detection, which monitors changes in the source code and can render the changes immediately in the web app on the browser. This greatly speeds up the development process, allowing for the addition of more features quickly. Our existing developers had a good knowledge of Angular. Angular allows for the simple integration of any JavaScript library that we wish to use. There is easy control of in application routing and module loading.

Look and Feel

We recognise the importance of the look and feel of a web application from both a user and a development point of view. From a user perspective, the web app must be easy to navigate. Names and links of different sections of the page should be clear. As much as possible, it should be obvious what the functionality of the different links and buttons are. For developers, a clear user interface is also important. When new functionality is developed, it makes the decision on where exactly to deploy

the functionality easier. For example, any notification functionality should be linked to the notification dropdown menu.

While this functionality and clarity is important, it is also important to make the user interface elegant and modern. Much thought and time was spent on the choice of an appropriate colour scheme and layout. Ultimately, it was decided that the Bootstrap-based theme “SB Admin” should be used. SB Admin uses the default Bootstrap 4 (Bootstrap, 2018) styles along with a variety of powerful plugins to create a convenient framework for creating admin panels, web apps and back-end dashboards. In addition, SB Admin provides a generic web template as well as an Angular-based template. This made integration with our application much easier. This is illustrated in Figure below.

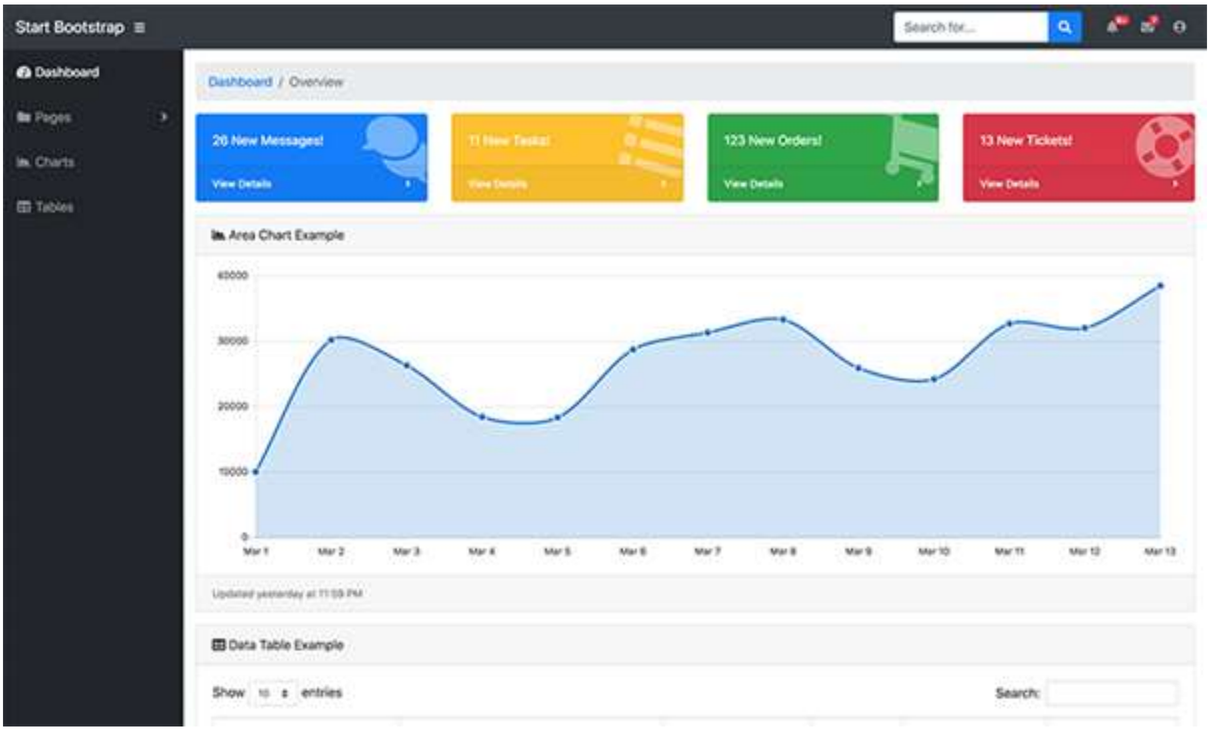

Figure 19: SB Admin template

Lazy Loading

One of the features that made Angular an attractive choice for the web app was lazy loading. In Angular, using components and modules, we can group related pieces of functionality together. Some pieces of functionality such as access to the user database or the initial dashboard are required at application start-up. However, some modules and components are not strictly required to load at start-up, for example, a very specific graphing module. In these cases, we can use lazy loading of modules. This means that the modules are only loaded when required on-demand when the user accesses certain parts of the web application. This is advantageous because it can drastically decrease the start-up time of an application, which can improve the user experience.

Routing in Angular

We are all familiar with how to navigate through a web page or the Internet in general, click a link and we are redirected to a different page. In Angular, the “router” imitates this behaviour. It interprets a browser URL as an instruction to navigate to a client-generated view. It supports the passing of parameters along to the specified view component in order for it to decide what content to present. Certain routes in a web site or web application can be “guarded” with programmatically defined

criteria. For example, a user cannot access a certain page if they are not logged in. Angular guards will be discussed more in 2.2. In addition, we can decide whether to handle the routing mechanism in angular centrally in one routing file or to distribute the responsibility for routing over several files. In this project, we have opted for the latter. The project consists of so many modules that it made sense to manage routing on a module-by-module basis.

3.9.2 Implementation choices

This section details the decisions that were made in relation to the implementation of the Prot-Dash web application. Several concepts will be explained and it will be made clear why we needed to conceive these concepts.

User Interface

To allow the user to view many different visualisations at the one time, widgets are used. The user can define as many widget as they wish. All of them will be displayed on the specific dashboard where the user defined them. It should be noted that the convenient user interface allows the selection of implemented Data Providers, Mapping Providers and View Providers (described below). As an additional feature, where possible, selection of non-compatible Providers is disabled to avoid user confusion. Widgets can quickly be moved, removed or resized. For convenience, it is possible to view a full screen version of the widget to allow more close examination of a visualisation. A widget can also be edited to alter the data being displayed or make some custom changes to the visualisation. A widget is illustrated in the following Figure.

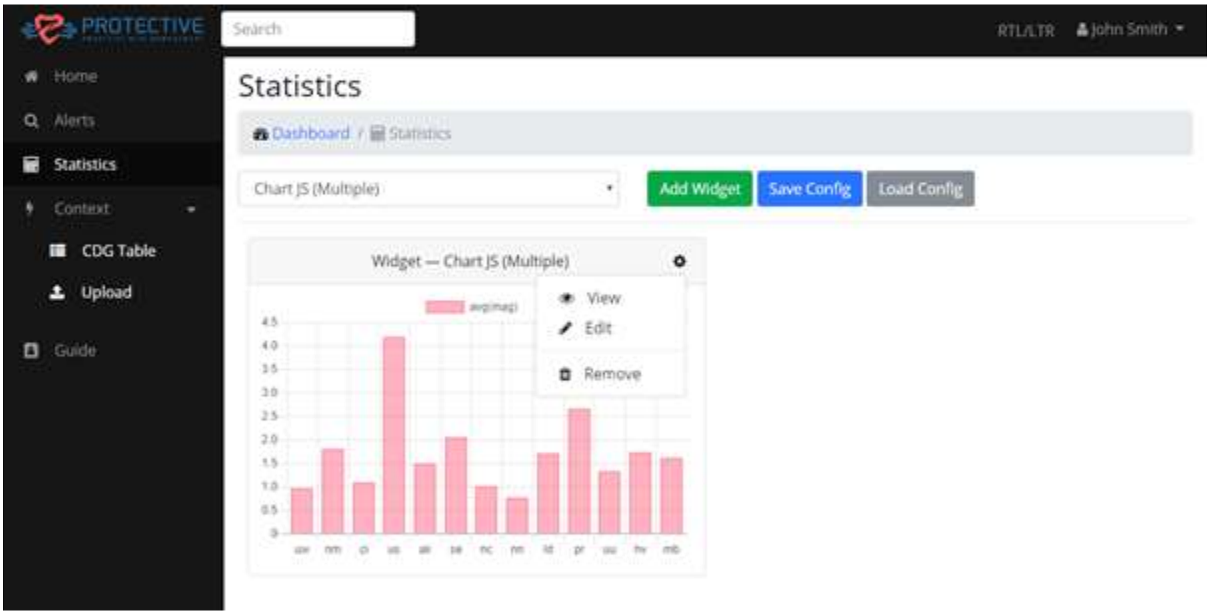

Figure 20: Widget

Prot-Dash Flexibility

Upon careful consideration, it was decided that one of the most important concerns when implementing Prot-Dash was that it was flexible. Flexible in terms of what functionality could be added by integrating already existing systems and libraries. If we implement the web app to only be able to get data from Rest APIs that return a specific format, then we limit the amount of data we have access to. Other data sources could be adapted by creating custom Rest APIs but, if we had to do this on a case-by-case basis, this would greatly increase development time.

Similarly, in terms of visualisation, if we limit the data to only one type of graph, then we are hiding insights from the user. Javascript has a great many visualisation libraries and we considered this in our implementation of Prot-Dash. We realised that not all of these libraries accept data in the same format and therefore had to take this into account in our implementation. The following paragraphs regarding to Data Provider, Mapping Provider and View Provider explain the three core concepts that allow Prot-Dash this flexibility. How these concepts fit into the web application is illustrated in Figure 21.

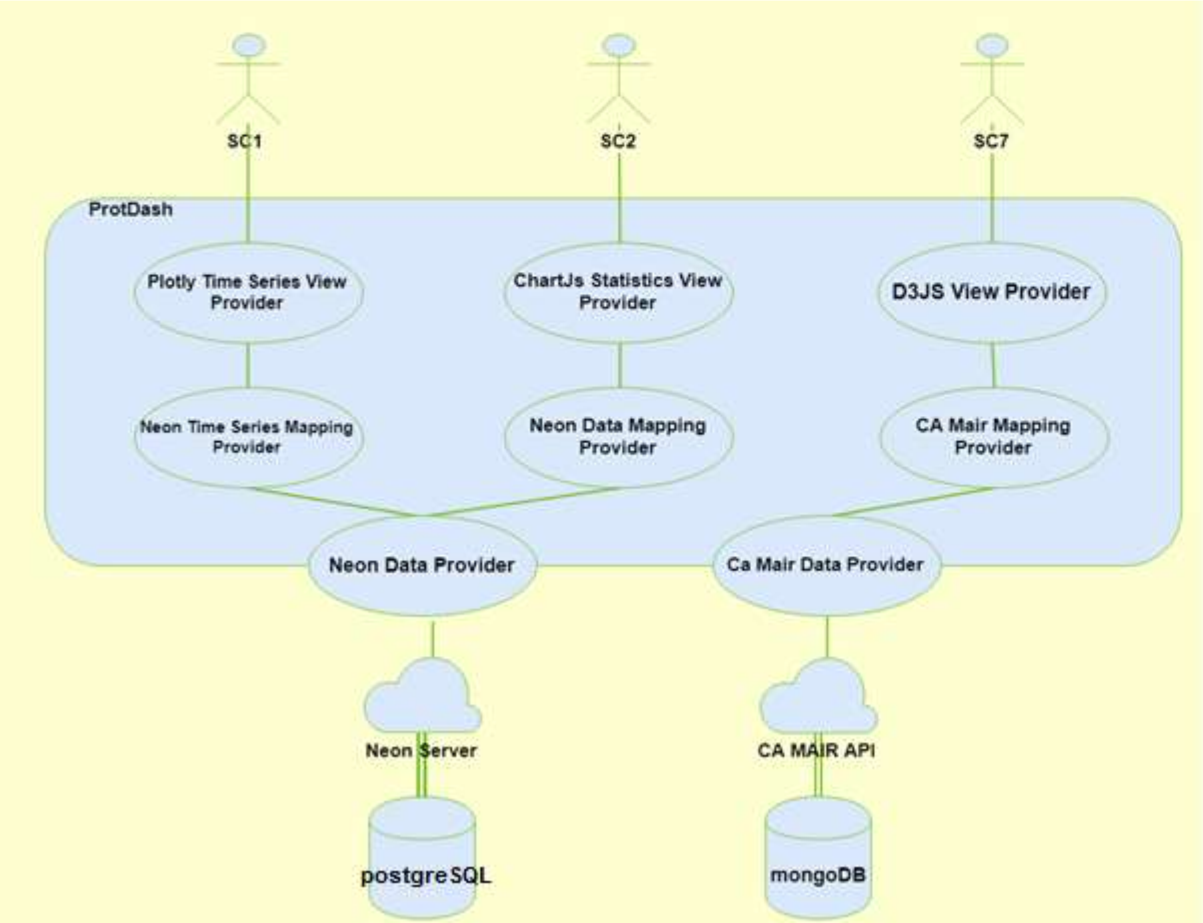

Figure 21: Prot-Dash Concept Mapping

Neon Server: Neon server is part of a larger project (Neon, 2018) that has the aim of providing a visualisation/data access framework for accessing and visualising many kinds of data easily. Neon server's Data Access API allows users to send a query to No-SQL databases using a SQL-like language. Neon does the "heavy lifting" by converting the query to a format that is understood by the target database. So, there is no need to create database-specific constructs. Neon allows this API to be accessed by a Javascript library or a restful endpoint. We use the former in our project. Neon server currently supports MongoDB, ElasticSearch and Spark SQL database back-ends.

Using the neon Data Access API Javascript library in our project, we created a Data Provider that accesses a MongoDB database. This database contains alert and security vulnerability information that is produced by other elements of the Protective project. The Neon Data Provider is configurable through the web application interface. Different collections and fields can be chosen within the UI, date ranges can be specified and data aggregations such as counts and averages can be chosen by the user.

Data Provider

The concept of the Data Provider is a generic object that can be adapted to provide data to the application from any data source, such as any kind of database, using Rest API calls, using specific query languages or other sources. The Data Provider in Prot-Dash is an interface that is used by any service or component in our Angular project that accesses some arbitrary data source.

All the interface requires is that any service or component that uses it has three different methods. The first method getDataType() retrieves the data type of the data that will be incoming from a specified data source. The second method, testDataProvider() serves to test both the availability and the functionality of a data source. For example, to see if data is returned in the correct format. The final method is the query() method, which makes the call to the data source and returns the data that can later be used in visualisations.

Any service or component that uses the Data Provider interface can add whatever other methods it needs to query or handle the incoming data, as long as it implements the methods describe above. Any new Data Provider that is created will be added as an option for the user to select on the web user interface. So, the user can easily choose between different data sources.

Mapping Provider

The basic idea of a Mapping Provider is to convert any data that is entering Prot-Dash from the data source data format to a format that can be used by whatever visualisation library the user wants to display data with. For example, the data source may provide data that is in raw JSON format but the graphing library may require an object with; date, time, value and weight properties. Any manipulation of the data in this regard takes place in the mapping provider.

In terms of programmatic implementation, the Mapping Provider is supplied as an interface that can be implemented by a service or component that needs to do conversion of data format. An input() method is used to pass data into the Mapping Provider. The supportsDataType() method helps to determine whether the Mapping Provider being used actually supports the data type coming from the data source. If the data type is supported, data is loaded into the Mapping Provider using the input() method. The supportsViewType() method is generally called after the format conversion has taken place and is used to determine whether the “mapped” data is supported by the type of visualisation the user wants to use. If the data type is supported, the output() function is called to pass the data to the visualisation object or View Provider.

View Provider

The final concept that will be described is the View Provider, which uses graphing, charting and illustration Javascript libraries to provide visualisations for the data that has been formatted by the Mapping Provider. This illustrates the flexibility and power of the Prot-Dash concepts. Potentially any graphing library can be used and there are many. Some examples; Flot, chart.js, D3.js, NVD3, Sigma.js or Cytoscape.js to name a few. Although not all of these libraries have been put into View Provider implementations yet, it can be easily done in the future because of the simple flexibility of the web application.

The code design of the View Provider is slightly different than the other Providers in that it doesn’t just use a View Provider interface, it also inherits from a Base View Provider. There was no need to define a blanket interface that required the interface user to define a lot of methods required by the interface because the majority of methods are common across all View Providers. Inheriting from the Base View Provider component meant much less repetition of code and easier implementation

overall. The inherited component provides many method implementations to do with discovering the correct Mapping and Data Providers, configuring the View Provider and displaying contextual error messages.

Viewing Libraries Used

Chart.js: Simple, clean and engaging HTML5 based JavaScript charts (chart.js, 2018). Chart.js is an easy way to include animated, interactive graphs on any website or web app for free. A View Provider for Chart.js has been created that uses all 7 available charts; line, bar, radar, doughnut, pie, polar area, bubble and scatter. The bar chart is shown in Figure 20. Using the edit option on the widget dropdown menu allows the user to switch between the different types of charts.js charts to experiment with which view is the best.

D3.js: D3.js is a highly configurable low level library that can be used to create a multitude of different visualisations (D3JS, 2018). In particular, we use d3.js to create a force directed graph to visualize the data received from CA MAIR (0). The force directed graph shows the nodes of the directed graph in a colour coded way with a legend. The links or lines between the nodes represent some relationship between those nodes. The graph is interactive. The nodes can be moved around to make examination of the graph detail easier. As there may be hundreds of nodes to a graph, pan and zoom functionality has been added to make it more convenient for the user to visualise. Hovering over a node or a link will allow cause a label to appear, which gives some details about the data. Hovering over a node will also trigger highlighting of every link and node that is connected, making viewing relationships easier. It is also possible to filter out/in categories of nodes by clicking on the relevant label/type in the graph legend. More detailed node data can be viewed by clicking on the edit option in the upper right corner of the widget and then clicking the View Provider tab. Detailed JSON data is available there on whatever node or link is clicked.

More sample D3JS visualisations have been provided here to show what D3JS is capable of but these implementations are not full.

Plotly: Built on top of d3.js and stack.gl, plotly.js is a high-level, declarative charting library. plotly.js ships with 20 chart types, including 3D charts, statistical graphs, and SVG maps (plotly, 2018). We use plotly for the visualisation of alert time series data from neon. A stacked bar chart is produced that can be used to show alert or vulnerability data grouped by type of attack or any other available statistic the user wants to group by. The time period, date and granularity of the data returned can be specified in the query. An interactive graph is produced, where a number of options are available. The time granularity can be changed, we can zoom or pan and hovering over the stacked bars details all the groups present in the bar.

Example Application UI implementation

CA MAIR API: The CA MAIR API is the part of the PROTECTIVE that provides context-awareness data from a backend MongoDB database. The data specifically has to do with assets (hardware, software, network…_) that have been uploaded to the database through CA MAIR. Several restful endpoints have been exposed by CA MAIR that allow easy querying the database to acquire information on specified assets.

Specifically, we create a data provider for CA MAIR to retrieve data based on the names of specific assets within the database. The ultimate aim of this Data Provider is to construct a directed graph with granular information on every single asset present and the relationships between those assets.

3.10 Connectors

In a PROTECTIVE system, a connector represents the way to send and receive data into and from the Warden Server component. So a connector is, in the case of a sending client, a piece of software that transforms data from the proprietary data format of one security tool (IDS, honeypot, network probe) into IDEA format and sends the data into Warden server. The current connectors, available for use with PROTECTIVE, are:

IntelMQ Connector

A connector for IntelMQ delivered as docker image. This connector reads data automatically from 3 sources:

- Malc0de
- Spamhaus
- Malware Domain List

With his customizations, it parses the events and convert it to IDEA format in order to get IDEA formatted events that warden-filer could pick and send to Warden without problems.

Kippo Connector

The Kippo connector (executable warden3-kippo-sender.py) is a one-shot script to send events from Kippo honeypot toward the Warden server directly.

The script warden3-kippo-sender.py does not run as a daemon, for regularly run, job scheduler cron must be used.

This connector is provided with a dummy data generator (gen_idea_kippo.py) that generates some dummy data in order to simulate the events coming from the real source.

Dionaea Connector

The Dionaea connector (executable warden3-dio-sender.py) is a one-shot script to send events from Dionaea honeypot toward the Warden server directly.

The script warden3-dio-sender.py does not run as a daemon, for regularly run, job scheduler cron must be used.

This connector is provided with a dummy data generator (gen_idea_dio.py) that generates some dummy data in order to simulate the events coming from the real source.

LaBrea Connector

The LaBrea connector (executable labrea-idea.py) is a daemon, meant for continuous watching of LaBrea log files and generation of IDEA format events of corresponding security events.

It needs to be run in correspondence with warden_filer daemon, which picks the resulting events up and feeds them to Warden Server.

Connector supports sliding window aggregation, so sets of connections with the same source are reported as one event (within aggregation window).

This connector is provided with a dummy data generator (gen_idea_labrea.py) that generates some dummy data in order to simulate the events coming from the real source.

Juniper SRX Connector

Juniper SRX connector (executable srx_connector.py) is a one-shot script that reads logfile (/var/log/srx.log), parse it to IDEA format and sends the generated events to Warden Server using warden-client.

The script srx_connector.py does not run as a daemon, for regularly run, job scheduler cron must be used.

This connector is provided with a dummy data generator (srx_generator.py) that generates some dummy logs in order to simulate the events coming from the real source.

McAfee Connector

McAfee connector (executable siem_connector.py) is a one-shot script that reads logfile (/var/log/siem.log), parse it to IDEA format and sends the generated events to Warden Server using warden-client.

The script siem_connector.py does not run as a daemon, for regularly run, job scheduler cron must be used.

This connector is provided with a dummy data generator (siem_generator.py) that generates some dummy logs in order to simulate the events coming from the real source.

Warden Parser

It parses any data source to idea files, at the moment only MySQL DAO is implemented. The code is extendable, so new DAO's can be implemented and added to the factory. It's implemented using PHP and MySQL, Apache is not necessary.

This connector is provided with a set of real logs in order to simulate the events coming from the real source.

FortiGate Connector

FortiGate connector (warden3-forti-sender.py) is a one-shot script that reads logfile (/var/log/forti.log), parse it to IDEA format and sends the generated events to Warden Server using warden-client.

The script warden3-forti-sender.py does not run as a daemon, for regularly run, job scheduler cron must be used.

This connector is provided with a set of real logs (fortigate-full.log) in order to simulate the events coming from the real source.

4 Implementation

4.1 Prerequisites

Hardware

PROTECTIVE has been verified in the following setups:

- Physical Machine with Ubuntu 16.04, 64GB RAM, 360GB HDD
- Virtual Machine OCI with Red Hat 4.8.5-28, 60GB RAM, 60GB HDD
- Virtual Machine KVM with Ubuntu Server 16.04.3 LTS, 12GB RAM, 40GB HDD
- Virtual Machine Openstack Rocky with Ubuntu 16.04 AMD64 LTS, 32GB RAM, 200GB HDD
- Virtual Machine VMWare ESXi with Ubuntu Server 16.04 LTS, 32GB RAM, 320GB HDD

Software

The machine (or VM) where PROTECTIVE Node will be installed must have nano, git, docker 17.120-ce+ and docker-compose 1.23.2+ installed.

We highly recommend using an Ubuntu 16.04 system. The account used to install Protective Node must be a member of the docker group. The account must also allow sudo without the need to enter a password. (This is the default setup for Ubuntu VMs)

4.2 Execution

For the PROTECTIVE system, code for all modules including the adapted Mentat and Warden are available in PROTECTIVE's GitLab https://gitlab.com/protective-h2020-eu/. An installation guide for the PROTECTIVE node is available at PROTECTIVE's Wiki https://gitlab.com/protective-h2020-eu/protective-node/wikis/home. The guide includes a detailed system description, instructions for installation of all the modules as well as instructions for setting up TI sharing with other peer PROTECTIVE nodes.

The modules available in PROTECTIVE System are:

For Warden:

https://gitlab.com/protective-h2020-eu/ti-s/Warden/Warden

For Mentat:

https://gitlab.com/protective-h2020-eu/cp/Mentat/Mentat

For the PROTECTIVE Node installation script:

https://gitlab.com/protective-h2020-eu/protective-node

For PROT-Dash source code:

https://gitlab.com/protective-h2020-eu/viz/ProtDash-Angular

For Keycloak source code:

https://gitlab.com/protective-h2020-eu/viz/keycloak

For the connectors, there is a folder for each one connector all the necessary to run them:

https://gitlab.com/protective-h2020-eu/ingestion/extraction

For the Context Awareness, the context awareness component software:

https://gitlab.com/protective-h2020-eu/enrichment/context-awareness

For the Meta-Alert Prioritisation (MAP), the software components are available in: https://gitlab.com/protective-h2020-eu/meta-alert-prioritisation

For the Correlation, the software components are available in the following repository:

https://gitlab.com/protective-h2020-eu/meta-alert-correlation

For the Trust Component, the software is available in the following repository:

https://gitlab.com/protective-h2020-eu/enrichment/TI-trust

For the Information Sharing Compliance module:

https://gitlab.com/protective-h2020-eu/information-sharing-compliance/runtime-monitoring

5 Annexes

5.1 Annex A: Warden Overview

5.1.1 Warden System

The Warden system is a third party product which was designed and developed by CESNET, a.l.e. NREN.

5.1.2 Warden Architecture

The architecture of the Warden system is that of the **client – server type**. The Warden system consists of a **server**, **receiving clients** and **sending clients** - see Figure 22 . The server, on request of receiving clients, distributes new (previously undistributed) events fed to the server by sending clients.

Each entity/network that wishes to feed data into the Warden system should have a so called **sending client**. Each entity/network that wishes to receive data from the Warden system should have a so called **receiving** client. The **server** (the centre) ensures the data reception and storage as well as the interface for the access to data stored. Data which the clients send into the centre are referred to as **events (alerts)**. Events are sent by the clients after authentication; the access to the centre is also authenticated. X.509 is used for the authentication.

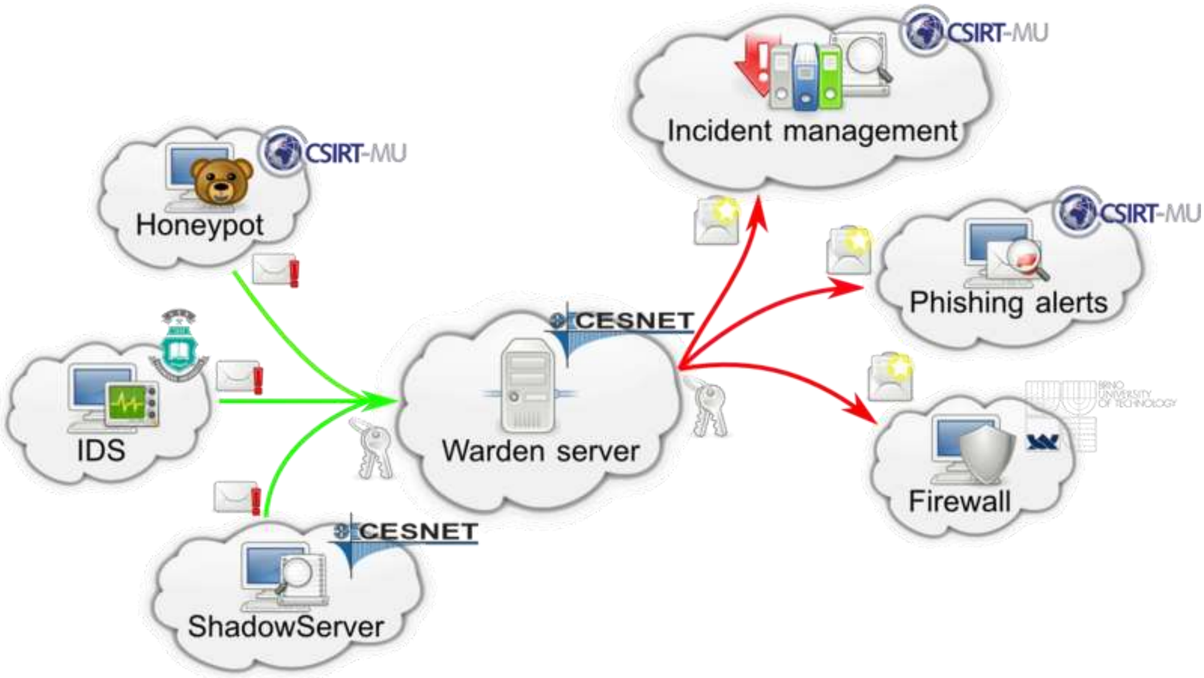

Figure 22: Warden System

Warden Server is server-side part of the software, the communication hub, allowing to publish detected events and download yet unprocessed ones. The events are exchanged in IDEA, flexible and descriptive event JavaScript Object Notation (JSON) serialized format. Warden 3 protocol is based on plain HTTPS queries with help of JSON, so the clients can be thin and simple.

Warden Server is Python/ *Web Server Gateway Interface* (WSGI) based, written primarily with Apache mod_wsgi in mind. Other WSGI servers/frameworks are not yet tested, so your mileage may vary. Authentication is X509 certificate (for machine or client identification) + shared secret (for client identification, where certificate does not suffice). The Python WSGI layer is run under venerable Apache web server.

Warden Clients can be multilanguage, as plain HTTPS and JSON is mature in many mainstream programming languages. Server is written in Python - mature language with consistent and coherent libraries and many skilled developers.

Warden uses flexible and descriptive event format, based on JSON. Warden 3 protocol is based on plain HTTPS queries with help of JSON.

Warden is a system for efficient sharing information about detected threats, available under 3-clause *Berkeley Software Distribution* (BSD) license. The latest versions of Warden server and client software (along with archive checksums) is available here: https://warden.cesnet.cz/en/downloads.

5.1.3 Concepts

PROTECTIVE project uses the term “alert”, Warden project uses the term “event”, both term describe the same thing. According to the “WP3 terminology dictionary” interim document (which will be also a part of the D3.1), alert is a notification that a specific attack has been directed at an organization’s information systems, with a known structure, composed of a list of attribute-value pairs.

Event description format

Event description format in Warden system is IDEA - Intrusion Detection Extensible Alert, flexible extensible format for security events, see: https://idea.cesnet.cz.

Event serial ID

Each received event gets assigned integer serial number. These numbers are sequential, so each recipient can keep track of the last event "id" it received and next time ask only for following events.

Authentication

In Warden 2 (original version of Warden), clients get authenticated by server certificate, however server certificate is usually same for the whole machine, so individual clients are differentiated only by telling its own name. However, client name is widely known, so this allows for client impersonation within one machine. Warden 3 slightly improves this schema by replacing client name in authentication phase by "secret", random string, shared among particular client and main server, which makes it harder to forge client identity (be it by mistake or intentional).

However, best solution for these cases is of course specific certificate for each particular client (which is also fully supported).

Client also has to have server *Certification Authority* (CA) certificate (or chain) at its disposal to be able to verify server authenticity.

Client name

Client names in Warden have hierarchy. Modelled after Java class names, client name is dot separated list of labels, with significance from left to right – leftmost denoting largest containing realm, rightmost denoting single entity.

Country.organisation.suborganizations.machine.local realm scheme akin to “org.example.csirt.northwest.honeypot.jabberwock" is strongly recommended. Label case is significant, label can contain only letters, numbers or underscore and must not start with number.

The reason is the possibility to filter incoming events based not only on particular client, or (for some recipients flawed) notion of "own" messages, but based on wider units.

HTTP/JSON API

Client must know the base URL of the Warden server. Warden 3 accepts queries on paths under base URL (which correspond to called method), with usual query string variable=data pairs separated by ampersand as arguments. Multivalues are specified by repeating same variable with each value several times.

https://warden.example.org/warden3getEvents?secret=PwD&cat=Abusive.Spam&cat=Fraud.Phishing

URL

Called method

Key/value pair

Multivalue

Method may expect bulk data (events to save, for example) - query then must be POST, with POST JSON data, formed appropriately as documented in particular method. If HTTPS call succeeds (200 OK), method returns JSON object containing requested data.

5.1.4 Design

Protocol

Warden RPC calls essentially consist of parametrised **event pull**, unconditional **event push** and service calls for getting information from server. Pull can be realised by standard bare HTTP call, however as HTTP notion of return data are general "documents", we will have to choose some way to serialise. In push direction, sending event data through HTTP GET parameters is impractical due to encoding concerns and size limits, so POST with the same serialisation format would be feasible. While we can consider eXtensible Markup Language (XML), there exists much lighter solution, which gained widespread recognition, is able to directly represent fundamental data structures from various programming languages, and is often used together with various HTTP technologies – JSON.

Format

Warden is based on IDEA data format, structured and extensible format, which already uses JSON as main representation. With IDEA Warden also get mature incident categorization (based on MkII) and expressive set of detector description tags for free.

Also, there are already tools in place for IDEA, which provide validation according to JSON schema definition, solving yet another requirement.

Filtering

Based on experience, we don't need overly complex filtering, Warden should serve mostly as reliable transport mechanism, not data-mining store or security event search engine. IDEA gives us notion of categories and detector tags, so we will allow for positive ("has category") and negative ("does not have category") filters on these fields. That will satisfy both use for searching by type (all portscans will have category *Recon.Scanning*) and for searching by detector type (if we want to get only confirmed attacks, we can filter out only detectors, based on successful attack – by for example *Honeypot* tag).

Organisational hierarchy filtering

We are still facing problem with notion of “own” events. Two administrators from one organisation may have their own reasons to either accept each others detectors data as “own” (they already have the data internally), or to understand each others detectors data as foreign (they want the data to arrive through Warden). As we cannot force any kind of rigid resolution onto them, we have to provide solution, which allows to project their notion onto the system and use it for filtering of “own” wanted/unwanted events.

Logical solution would be using hierarchy of Domain Name Service (DNS) names. However, keeping more complicated structure in DNS servers gets inconvenient very quickly, and also may unwillingly disclose addresses of the detectors or honeypots. As we do not need complete distributed name infrastructure, we can use hierarchical IDEA Node names as the base and allow organisations to define identifiers inner structure themselves. So, modelled after Java class names, client name is dot separated list of labels, with significance from left to right – leftmost denoting largest containing realm, rightmost denoting single entity. So if we have name realm scheme akin to "org.example.csirt.honey2", we can allow to filter based on prefixes and it is than completely responsibility of organisation, what hierarchy and names it will use and how it will filter incoming events.

We can also allow both positive and negative filters.

Bulk send/receive

Pull API is able to provide client with requested number of events on one call, however here server is at command at maximum limit of events it is willing to send. If we allow bulk transfer for push API, server may receive arbitrary number of events – even very large number – in one call. As server has to balance throughput and responsiveness, it also has to do some limiting. We will thus let server to present client with limit constants (for both directions) in initial handshake communication, and also in error message structures, should the client overflow these constants

Authentication

To mitigate possibility of impersonation among clients on one machine, clients will have to supplement shared secret (instead of their publicly known name) during queries. As the connection is always encrypted over HTTPS and shared secret is distributed only once on client registration over secure channel, there is no need to complicate things with additional encryption or handshake scheme. However note that this mechanism is only for transition phase to specifically tailored certificates, which will contain client (not only machine) identifier directly.

Clients will also have to have server authority certificate (or chain) at their disposal to be able to verify server authenticity.

State

At the server, each event gets assigned integer serial number. These numbers are sequential, so we can keep track of the last event "id" each client have received and next time provide him only with yet unseen events.

Server will also keep state of the last downloaded event for each client, thus freeing clients from necessity to keep permanent state themselves – however clients are free to provide their own notion of state id for each query and saving it on their own, should the need arise – for example while using more sophisticated filtering schemas.

Logging/Debugging

Sometimes administrators need to debug problems, which arise only when using specific client or specific setup. Errors like these are hard to hunt both from client and server side. We should try provide administrators with enough information and tools to simplify hunting of problems like these.

Usual attempt is to use logging. Good lesson to learn comes from Postfix Mail Transfer Agent (MTA). Each mail message, entering the system, receives sufficiently unique identifier, which gets propagated to all log messages, appears in Simple Mail Transfer Protocol (SMTP) communication and ends up even in *Received* headers of the mail messages. In Warden, each query can also acquire unique identifier to get written into all related log messages at various parts of the system, and this id can get returned in the case of errors.

Also, server should provide some call to acquire basic information about server, its capabilities and limits.

5.1.5 HTTP API Design

PROTECTIVE can identify two three classes of transferred data – structured event data (transferred both directions), error explanations (only from server to client) and query modification arguments (only from client to server).

Considering modification arguments, such as authentication tokens, filtering and first event ID, the well understood and widely used notion of URL parameters suits well – both sides know the type of the value, so we only need to transfer key/value pairs of strings. Repeated arguments can easily mimic multivalues/arrays.

For structured data in push direction POST data can be used and for pull direction we can send resulting data directly. However, we will have to settle for structure.

The examples are provided as calls to command line HTTP client utility *cURL*[1], which also shows that by using this design we are able to access server methods even without client library, which is very useful for debugging, and can be also used as a base for very lightweight clients.

Error handling

If HTTPS call succeeds (200 OK), method returns JSON object containing requested data.

Should the call fail, server returns HTTP status code, together with JSON object, describing the errors (there may be multiple ones, especially when sending events). The keys of the object, which may be available, are:

- *method* – name of the called method
- *req_id* – unique identifier or the request (for troubleshooting, Warden administrator can also uniquely identify related log lines)
- *errors* – always present list of JSON objects, which contain:
    - *error* – HTTP status code
    - *events* – list if indices of events, affected by this particular error. If there is error object without *events* key, caller must consider all events affected

1 Available:

  - *message* – human readable error description

Other context dependent fields may appear, see particular method description.

Client errors (4xx) are considered permanent – client must not try to send same event again as it will get always rejected – client administrator will need to inspect logs and rectify the cause.

Server errors (5xx) may be considered by client as temporary and client is advised to try again after reasonable recess.

Common arguments

- *secret* – shared secret, assigned to client during registration
- *client* – client name, optional, can be used to mimic Warden 2 authentication behaviour if explicitly allowed for this client by server administrator

getEvents method

Fetches outstanding events, that means events with *id* higher, than last downloaded, from server.

Arguments

- *count* – number of requested events
- *id* – starting serial number requested, id of all received events will be greater
- *cat*, *nocat* – selects only events with categories, which are/are not present in the event Category field (mutually exclusive)
- *group*, *nogroup* – selects only events originated/not originated from this realms and/or client names, as denoted in the event Node.Name field (mutually exclusive)
- *tag*, *notag* – selects only events with/without this client description tags, as denoted in the event Node.Type field (mutually exclusive)

Returns

- *lastid* – serial number of the last received event
- *events* – array of IDEA events

Example

```
$ curl \
 --key key.pem \
 --cert cert.pem \
 --cacert ca.pem \
 --request POST \
 \
"https://warden.example.org/getEvents?\
secret=SeCrEt\
&count=1\
&nogroup=org.example\
&cat=Abusive.Spam\
&cat=Fraud.Phishing"

{"lastid": 581,
 "events": [{
 "Format": "IDEA0",
 "DetectTime": "2015-02-03T09:55:21.563638Z",
 "Target": [{"URL": ["http://example.com/"]}],
```

```
 "Category": ["Fraud.Phishing"],
 "Note": "Example event"}]}
```

sendEvents method

Uploads events to server.

Arguments

- POST data – JSON array of Idea events

Returns

Object with number of saved messages in *saved* attribute.

Example:

```
$ eventid=$RANDOM$RANDOM$RANDOM$RANDOM$RANDOM
$ detecttime=$(\
 date --rfc-3339=seconds|tr " " "T")
$ client="cz.example.warden.test"
$ printf '
[
 {
 "Format": "IDEA0",
 "ID": "%s",
 "DetectTime": "%s",
 "Category": ["Test"],
 "Node": [{"Name": "%s"}]
 }
]' $eventid $detecttime $client |\
curl \
    --key $keyfile \
    --cert $certfile \
    --cacert $cafile \
    --request POST \
    --data-binary "@-" \
    "https://warden.example.org/sendEvents?"\
    "client=$client&secret=SeCrEt"

{"saved":1}
```

Example with error:

```
$ curl \
 --key $keyfile \
 --cert $certfile \
 --cacert $cafile \
 --connect-timeout 3 \
 --request POST \
 --data-binary '[{"Format":'\
 '"IDEA0","ID":"ASDF","Category":[],'\
 '"DetectTime":"asdf"}]' \
 "https://warden.example.org/sendEvents?"\
 "client=cz.example.warden.test&secret=SeCrEt"

{"errors":
 [
 {"message": "Validation error:
key \"DetectTime\", value \"asdf\", expected
- RFC3339 timestamp.",
 "events": [0],
 "error": 460
 }
 ],
 "method": "sendEvents",
 "req_id": 3726454025
}
```

getInfo method

Provides client with basic server information.

Returns

- *version* – Warden server version string
- *description* – server greeting
- *send_events_limit* – sendEvents will be rejected if client sends more events in one call
- *get_events_limit* – getEvents will return at most that much events

Example

```
$ curl \
 --key key.pem \
 --cert cert.pem \
 --cacert ca.pem \
 --connect-timeout 3 \
 --request POST \
 "https://warden.example.org/getInfo?secret=SeCrEt"

{"version": "3.0-beta1",
 "send_events_limit": 500,
 "get_events_limit": 1000,
 "description": "Warden 3 server"}
```

Each pull query is based on *id*, provided by client, signalling which events it has already received. Clients may also require to filter events according to category, detector tags and trailing part of detector name (*"realm"*).

Push queries are nothing special, they will just have to update all potential auxiliary structures accordingly – but the backend database engine must provide enough locking granularity to be able to cope with continuous stream of writes along with continuous stream of reads.

Each query has to be authenticated by client shared secret and/or client name, and we should be able to differentiate between clients, which are allowed to send, clients, which are allowed only to receive, and new, unverified clients, which are able to send only events marked with specific "Test" category.

5.1.6 Python Wrapper API

Python API tries to abstract from raw HTTPS/URL/JSON details. User instantiates *Client* class with necessary settings (certificates, secret, client name, logging, limits ...) and then uses its method to access server.

Client constructor

```
wclient = warden.Client(
    url,
    certfile=None,
    keyfile=None,
    cafile=None,
    timeout=60,
    retry=3,
    pause=5,
    get_events_limit=6000,
    send_events_limit=500,
    errlog={},
    syslog=None,
    filelog=None,
    idstore=None,
```

```
    name="org.example.warden_client",
    secret=None)
```

Arguments

- *url* – Warden server base URL
- *certfile*, *keyfile*, *cafile* – paths to X509 material
- *timeout* – network timeout value in seconds
- *retry* – number retries on transitional errors during sending events
- *pause* – wait time in seconds between transitional error retries
- *get_events_limit* – maximum number of events to receive (note that server may have its own notion)
- *send_events_limit* – when sending, event lists will be split and sent by chunks of at most this size (note that used value will get adjusted according to the limit reported by server)
- *errlog* – stderr logging configuration dict
    - *level* – most verbose loglevel to log
- *syslog* – syslog logging configuration dict
    - *level* – most verbose loglevel to log
    - *socket* – syslog socket path (defaults to "/dev/log")
    - *facility* – syslog facility (defaults to "local7")
- *filelog* – file logging configuration dict
    - *level* – most verbose loglevel to log
    - *file* – path to log file
- *idstore* – path to simple text file, in which last received event ID gets stored. If None, server notion is used
- *name* – client name
- *secret* – authentication secret

Returns

Client object, which provides methods, exposing and simplifying Warden HTTP API.

Configuration file helper

```
warden.read_cfg(cfgfile)
```

read_cfg allows for object to get initialized from JSON like configuration file. It's essentially JSON, but full line comments, starting with "#" or "//", are allowed. *read_cfg* reads the configuration file and returns dict suitable for passing as *Client* constructor arguments.

Arguments

- *cfgfile* – path to JSON configuration file, relative to base script position

Returns

Dict, prepared from read JSON data.

Example

```
wclient = warden.Client(
 **warden.read_cfg("warden_client.cfg"))
```

warden.Client.getEvents

```
wclient.getEvents(
    id=None,
    idstore=None,
    count=1,
    cat=None, nocat=None,
    tag=None, notag=None,
    group=None, nogroup=None)
```

Gets outstanding events from server by getEvents HTTP call.

Arguments

- *id* – can be used to explicitly override value from *idstore* file; corresponds to *id* in HTTP API
- *idstore* – can be used to explicitly override idstore for this request
- *count*, *cat*, *nocat*, *group*, *nogroup*, *tag*, *notag* – correspond to their HTTP API counterparts

Returns

List of IDEA events from queue greater than *id.*

warden.Client.sendEvents

```
wclient.sendEvents(
 self, events=[], retry=None, pause=None)
```

Arguments

- *events* – list of events to be sent to server
- *retry, pause* – use this values just for this call instead of the value from constructor

Returns

Dict with number of sent events under "saved" key.

Notes

Events list length is limited only by available resources, *sendEvents* will split it and send separately in at most *send_events_limit* long chunks (however note that *sendEvents* will also need additional memory for its internal data structures).

Server errors (5xx) are considered transitional and sendEvents will do retry number of attempts to deliver corresponding events, delayed by pause seconds.

Should the call fail because of errors, particular errors may contain "events" list. Values of the list are then indexes into POST data array. If no "events" list is present, all events attempted to send must be considered as failed (with this particular error).

Errors may also contain event IDs from Idea messages in "events_id" list.

This is primarily for logging – client administrator may identify offending messages by stable identifiers.

warden.Client.getInfo

```
wclient.getInfo()
```

Returns dictionary of server related information from *getInfo* call.

Error class

```
Error(
    message,
    logger=None,
    error=None,
    prio="error",
    method=None,
    req_id=None,
    detail=None,
    exc=None)
```

Class, which gets returned in case of client or server error. Caller can test whether it received data or error by checking:

```
isinstance(res, Error).
```

However if he does not want to deal with errors altogether, this error object also returns *False* value if used in *Bool* context and acts as an empty iterator – in following examples *do_stuff()* is not evaluated:

```
if res:
    do_stuff(res)

for e in res:
    do_stuff(e)
```

str(Error_Instance) outputs formatted error, *info_str()* and *debug_str()* output increasingly more detailed info.

5.2 Annex B: Mentat Overview

Mentat *is a distributed modular SIEM (Security Information Management System) developed by CESNET, a. l. e. and operated in CESNET e-infrastructure. Mentat is designed to monitor networks of all sizes. Its architecture enables reception, storage, analysis, processing and response to a great volume of security incidents originating from various sources, such as honeypots, network probes, log analysers, third party detection services, etc. Although the source code has not yet been made publicly available, the Mentat system has been developed as an open-source project.*

5.2.1 Mentat architecture overview

The Mentat system has been designed as a distributed modular system with an emphasis on its easy extendability and scalability – see Figure 23 and Figure 24. The core of the system reflects the architecture of MTA system Postfix. It consists of many simple modules (daemons), each of which responsible for performing a particular task. This approach enables smooth parallelization and extendability.

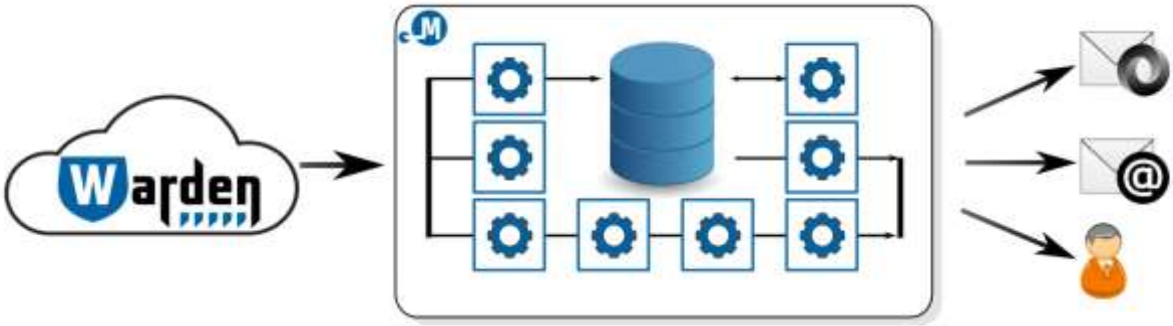

Figure 23: Mentat Architecture

All modules use the same core service framework, thus making implementing new modules an easy task. The core framework is responsible for dealing with all common tasks like:

- Configuration loading and validation;
- Daamonisation;
- Log initialization;
- Database abstraction layer;
- Message queue abstraction layer;
- Abstract layer for working with IDEA messages;
- Statistical data processing;
- WHOIS queries, DNS resolving, GeoIP resolving;
- Email report formatting and distribution.

Currently there are two types of Mentat modules:

- real-time processing modules (daemons, always running);
- post-processing modules (scripts, periodically launched).

All continuously running modules (deamons) operate as ‘pipes’, i.e. the message enters on one side, the deamon performs relevant operations and processing and then the message reappears on the other side. To facilitate message exchange between individual deamons, like in MTA Postfix, the message queues implemented by means of files and directories are used. When implementing a new deamon, one only needs to configure the processing; everything else is provided for automatically, including the selection of a message from the queue and subsequent upload into the queue of another deamon in the processing chain.

All periodically executed modules (scripts) usually perform a database query and then process batch of messages at once.

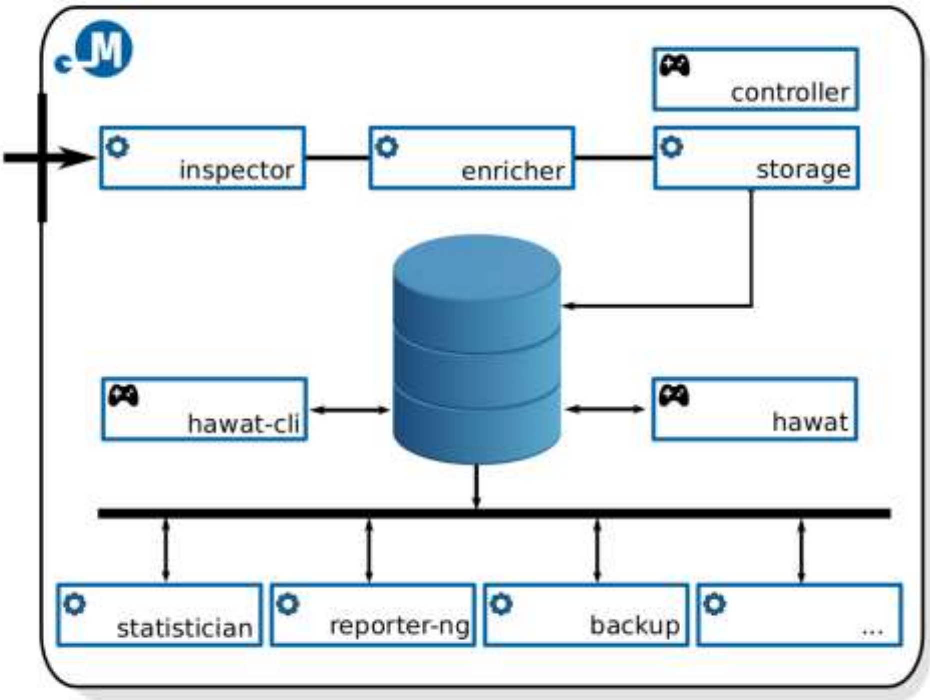

Figure 24: Mentat Node

Mentat development

Current version of Mentat is written in Perl and Python. New version is being (re)implemented in Python and all new code based on Mentat framework are and should be written in Python. Perl modules are obsolete and will be continually rewritten into Python and removed from Perl repositories.

Due to the design of Mentat system, modules may be written in any language, it is just necessary to take care of everything that is being taken care of by existing framework. Mentat real time modules communicate using filesystem directory queues, so you can simply (atomically) drop new message into the queue and it will be handled. You may use existing framework for implementing custom modules, there is however no tutorial or how to and you just have to roll up your sleeves, dig into the source code and use existing modules as blueprints for implementing new ones.

5.2.2 Mentat daemons

Mentat-inspector:

This real-time message processing module enables processing of IDEA messages based on the result of given filtering expression. There is a number of actions, which can be performed on the message in case the filtering expression evaluates as true. Currently following actions are supported:

- tag - Tag message with given static string;
- set - Tag message with the result of given expression;
- drop - Drop message from processing (filter out);
- dispatch - Dispatch (move) message to different processing queue;
- duplicate - Duplicate (copy) message to different processing queue;

- report - Report message to given email address with given subject;
- log - Log message to log file.

Mentat-enricher:

This real-time message processing module enables the enrichment of incoming messages with additional information. Currently only the resolving of target abuse contact for automated message reporting is supported. However, additional enrichment plugins can be implemented and added as necessary and the development version of this module already supports dynamical enrichment plugins and GeoIP resolving plugin is already implemented.

Mentat-storage:

This real-time message processing module is quite simple and enables storing of incoming messages into customizable MongoDB collection. It also takes care of all necessary datatype conversions and that is basically it.

5.2.3 Mentat Script-modules

Mentat-statistician:

This post-processing module enables statistical processing of IDEA messages over a given self-defined period. At present, the feature is preset to five-minute intervals. For each of these intervals, the module calculates the count of events according to detector type, event type, IP address etc. These statistical reports are stored in a separate database and can later support an overview of system's operation, provide underlying data for other statistical reports or for the creation of dictionaries for a web interface.

Mentat-reporter-ng:

This post-processing module enables distribution of periodical message reports directly to end abuse contacts of responsible network administrators. All messages in certain time period are aggregated according to event type and target abuse contact and based on the event severity and custom configuration of reporting algorithm an email report may be generated and sent directly to responsible administrators. Some of the messages might be filtered out from reporting using filters.

This post-processing module enables distribution of periodical message reports directly to end abuse contacts of responsible network administrators. All messages in certain time period are aggregated according to event type, event severity and target abuse contact and based on the custom configuration of reporting algorithm an email report may be generated and sent directly to responsible administrators.

The target abuse contact email addresses are already present in every message thanks to the Mentat-enricher component, which greatly speeds up the reporting process. The email information comes from from RIPE`s whois service.

Currently the reporter supports four severity levels (low, medium, high, critical) and for each of these levels reporting algorithm accepts period, threshold and relapse configurations. Period is a time interval, for which to perform aggregation. Threshold is a time interval, for which to withhold reporting of same events (same source IP address and event type) and thus lower the number of repeated reports. Relapse is a heuristic to detect successful or unsuccessful issue resolving and it will trigger reporting of message previously withheld during threshold time interval. The reporting is performed separately for each of these severities and the period/threshold/relapse configuration can be different, so that the less severe events can get more aggregated and more severe events reported

more quickly.

Additionally some of the messages might be filtered out from reporting using preconfigured custom filters and thus removing any known false positives from reporting.

Mentat-briefer:

This module is somewhat similar to the above described reporter, however reports generated by this module are more statistical and targeted for system administrator. They provide periodical summary on system status, performance and reports sent.

5.2.4 Mentat utility script modules

These modules provide management tools for the administrator of Mentat system.

Mentat-controller:

Configurable script enabling to control (start, stop, restart) all required daemon modules on a given server.

Mentat-backup:

Configurable script enabling periodical database backups. At present, a full backup of system collections (users, groups ...) is created once a day while event collections are backed up incrementally.

Mentat-cleanup:

Configurable script enabling periodical database and filesystem cache cleanup and responsible for data retention process.

Mentat-precache:

Configurable script enabling data caching, in particular of various dictionaries for web interface

Hawat-registry:

Script module enabling data synchronisation between Registry and Mentat’s system database. It synchronises abuse groups and address blocks assigned to them

5.3 Annex C: IDEA Format in PROTECTIVE

The following table shows the IDEA fields that are currently being used in PROTECTIVE during ingestion.

Field	Description
id	Unique message identifier
detecttime	Timestamp of the moment of detection of event (not necessarily time of the event taking place)
category	Category of event
description	Short free text human readable description
source_ip	IP addresses of this source
target_ip	IP addresses of this target
source_port	Ports of this source
target_port	Ports of this target
source_type	Type of this source
target_type	Type of this target
protocol	Protocols, concerning connections from/to this source/target
node_name	Name of the detector, which must be reasonably unique, however still bear some meaningful sense. Usually denotes hierarchy of organisational units which detector belongs to and its own name
node_type	Tag, describing various facets of the detector
node_software	The name of the detection software (optionally including version)
cesnet_storagetime	Timestamp of the moment of the moment the event was stored into database
cesnet_resolvedabuses	Abuse contacts related to any alert source
cesnet_eventclass	Event class determined by inspection
cesnet_eventseverity	Event severity determined by inspection
cesnet_inspectionerrors	List of event peculiarities found during inspection
event	Full event in binary format

Table 1: Ingestion IDEA fields

The following table shows the IDEA fields that are added to a PROTECTIVE alert during enrichment.

Field	Description
passive_dns	Result of the DNS lookup of all Source/IP addresses
entity_reputation	Future Misbehavior Probability (FMP) score fetched using the NERD Client.
score	The final trust score given to an alert as computed from the quality, certainty and source trustworthiness of the alert.
completness	The measurement of how much relevant information is reported by an alert. This measurement is at its maximum when all expected fields in the IDEA scheme definition of an alert are present. Scheme attributes (e.g., source IP address, category, node type) set as important in the configuration of the trust module increase the completeness value if they are present in the alert. For each missing field, this measurement is reduced.
alert_freshness	A measure of how recent an alert is. The "DetectTime" field of the IDEA scheme is used to convert the number of elapsed hours to a value between zero and one.

source_relevance	The confidence we have of an alert being correct and accurate. The confidence of different detectors is configurable. In the default configuration, Honeypots are assigned the most confidence while anomaly detectors the least.
quality	The alert's quality is a weighted mean of the completeness and the alert's freshness.
certainty	A measure of how certain an alert is. For example, Honyepots are expected to be more certain than anomaly detectors. The more certain an alert is, the more relevant it is for the calculation of an alert's trust score.
source_trustworthiness	A value that indicates how much we trust the organization that issued the alert. The alert's organization is determined by the prefix of the sensor's hostname. The final trust scores of previous alerts are persistently stored per organization and are used to calculate the current trustworthiness.
ip_recurrence	Number of times an IP address has been seen as the source of alerts.

Table 2: Enrichment IDEA fields