CESNET Technical Report 04/2013

Expert system Mentat

Jan Mach

Received 9.12.2013

Abstract

This paper describes the architecture, design and implementation of the expert system referred to as Mentat. It is a distributed system that consolidates gathering, storage and processing of intrusion alerts we receive both from our own detection systems and honeypots and from some third party services such as Shadowserver and Team Cymru.

Keywords

Security, SIEM, IDMEF, Perl

Introduction

Most network operators care about their network and are interested in what is going on in it. They usually apply some proactive methods and operate some IDSs (Intrusion Detection Systems) or honeypots to ensure their network is more secure for their users. They usually also have a CSIRT/CERT team the members of which are responsible for keeping track of information coming from various sources, assessing it and reporting problems to be dealt with.

We are in the same position. In the CESNET2 network, we operate several intrusion detection systems. Besides that, we receive alerts about problems in our network from third party services such as ShadowServer[1] or Team Cymru[2]. At present, every detection sys-

tem has a reporting tool of its own, own data format and own persistent storage. As a result, it is not possible to correlate, aggregate or search through information from various sources easily, or to generate event and processing statistics easily. On top of that, we have a CSIRT team the members of which use OTRS[3] ticket tracking system to handle incidents. It is rather time consuming to look for relevant information manually in multiple places. Thus, our biggest motivation for developing the Mentat system to consolidate event sources, event persistent storage and event processing.

The main goals of the Mentat system can be summarized as follows:

1. Development of a system for consolidation of intrusion detection information
2. The system should be able to receive, store and process information from various sources (IDS, honeypot, third party service, ...), i.e. variable data format
3. Persistent event storage should be fast and ensure great performance
4. The system should be able of both real-time and post event processing; this processing must be uniform for data from all sources
5. The system should be able to perform automatic actions based on fulfilling specific conditions
6. The system should provide user interface for searching persistent event storage using reasonably complex queries
7. The system is intended to be aid for CSIRT team members
8. System's key design features:
    1. Scalability (distributed architecture)
    2. Performance
    3. Transparency, configurability (no black box)
    4. Extendability (easy development of custom system components)
    5. Security, Integrity

At first we looked for some existing solution, that would satisfy all our needs. All closed source or proprietary solutions were excluded immediately as we wanted an open solution. We came across a few promising possibilities including Prelude[4], OSSEC[5] and OSSIM[6]. After considering advantages and disadvantages of the above systems we decided to test Prelude IDS. Although it is a very solid piece of software, we decided to abandon this approach aftex six months of testing and develop own solution that would satisfy our needs even better. This was mainly because of database and user interface performance issues as we were not able to obtain all requested information from a vast amount of data. In addition, the database and interface were not fast enough for our needs.

Design

System architecture

Based on the requirements and key features mentioned in the previous chapter, we developed a system called Mentat. Figure 1 depicts the system's design and architecture, inspired by Prelude IDS and Postfix[7] systems. It is designed as a hierarchical and distributed system composed of many "simple" one-task components. This approach has many benefits, including:

1. Scalability – The whole system may run on a single host, or it may be distributed between an arbitrary number of hosts.
2. Performance – Many simple one-task components make a good use of hosts computation resources due to natural paralelization
3. Security – Every component can run with strictly restricted permissions
4. Robustness – A failure of a single component does not cause the failure of the whole system
5. Messages (or events) may be retransmitted through a chain of components to multiple destinations; the components can be connected with arbitrary complexity.

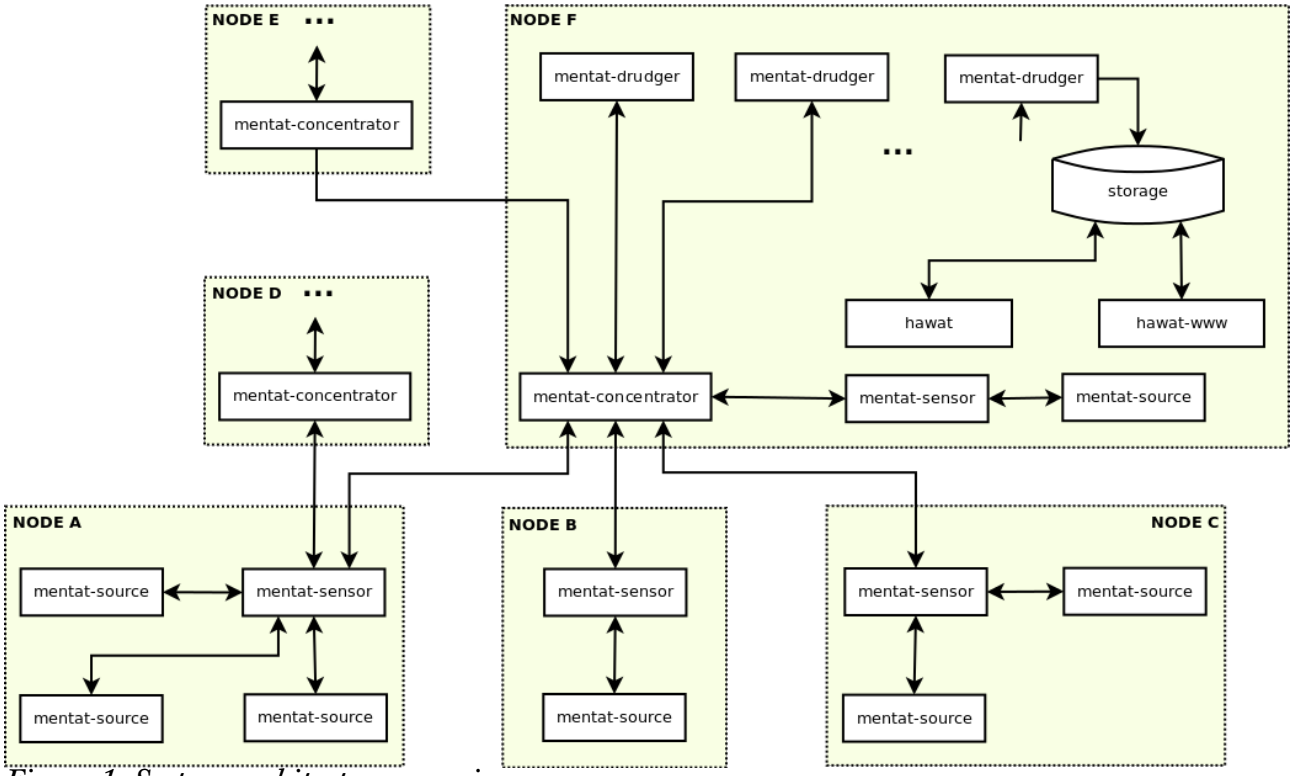

Figure 1: System architecture overview

Currently, there are five types (or groups) of the components:

1. Source
2. Sensor

3. Concentrator
4. Drudger
5. Hawat

The lowest level of the architecture consists of *source* components. These components are responsible for feeding messages into the system. The methods of operation of these components can vary substantially. They can detect events directly, for example by regular expression based parsing of log files, and generate the messages themselves. Or they can process third party messages and convert them into the internal data format. Alternatively they can act as simple pickup daemons: periodically picking up messages from third party systems contained in files in a designated monitored file system directory.

Every *source* can be connected to exactly one *sensor* or *concentrator* component. In small and less demanding setups, it is more efficient to connect *source* directly to the *concentrator*, but *sensors* are necessary when additional preprocessing or a more complex setup is needed. *Sensors* are basically local host *concentrators* (see below) responsible for gathering events from all local *sources*. This may seem like unnecessary overhead and bottleneck but as mentioned before, *sensors* are intended to perform additional event preprocessing beyond the scope of the simple *source*. This way, the preprocessing is configured only in one place and is done for all events from all *sources*. This preprocessing may be anything that is needed: message syntax validation, thresholding, message firewalling, cleaning or blocking from untrustworthy *sources*, message normalization or data conversions, data (de)anonymization, message aggregation to lower the load on the next system components, event sampling, etc. Another benefit of *sensor* is the existence of a single connection between the *sensor* – *concentrator* pair which may make a great difference in setups with many *sources* where could be thousands of connections opened to the central *concentrator*. And finally, sensors are capable of transmitting events to more than one *concentrator*.

Concentrators are the heart of the event transporting chain and work as filtering and dispatching units. They receive events from multiple remote or local *sources* and *sensors* and apply the filtering rules to dispatch the matching messages either to an arbitrary number of following *concentrators*, or to the local processing components called *drudgers* (see below). Besides that, they are responsible for component authentication and authorization.

Drudger components are doing all the hard work. They are responsible for event processing as such. The processing can be anything a Mentat user can come up with: storing the events to the persistent database storage, event correlation, statistical analysis, heuristics, event reporting, summary report generation or automatic actions such as blacklisting, firewalling, etc., i.e anything that is needed. All the processing is done in real-time which gives the system a great reaction time.

The final component of the system is a GUI called *Hawat*. Currently, this GUI can only be used to search the event database. Soon however it will be able to generate the reports, processing statistics and charts or view the system component structure as well as manage the event database.

Component architecture

Now let us go into more detail and focus on the internal architecture of the system compo-

nents. The design is the same for all types of components except the Hawat GUI which is the web interface written in Catalyst MVC Framework [7]. The component design is displayed on Figure 2.

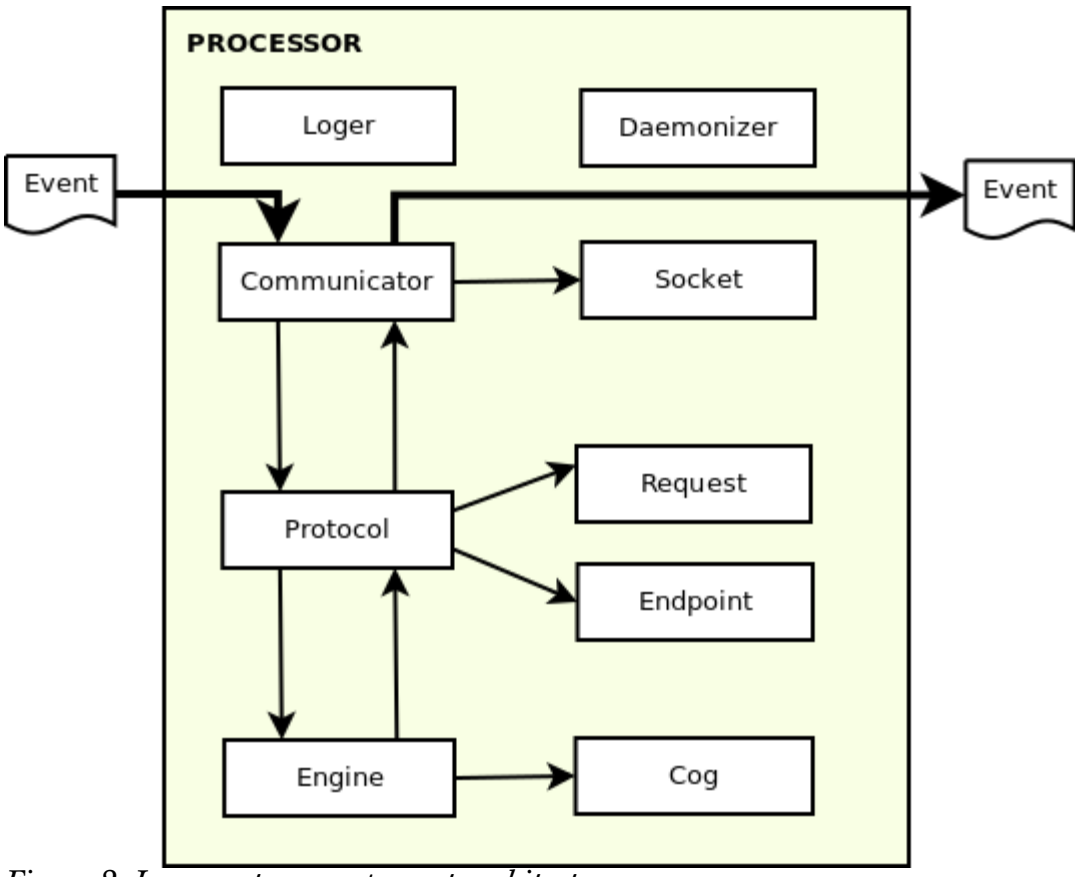

Figure 2: Inner system component architecture

Basically, all components work as simple pipes: messages come in from one end, are processed in certain way and finally are released at the other end. The only two slight modifications of this concept are the source components and some of the drudger components. The former need no input as they generate/feed own messages. The latter have no output (for example a component for storing events into a persistent storage).

A *processor* is a so called front controller and its purpose is to initialize the application environment, setup all subcomponents and services (logging); optionally also to daemonize the application and enter the run loop. The main application logic is contained within three processing layers: *communicator*, *protocol* and *engine*, and other helper objects. This design was inspired by TCP/IP stack, with each layer serving the higher layer and using services of the lower layer. The c*ommunicator* layer is responsible for direct communication and socket manipulation. It receives the incoming data and sends out awaiting data through an arbitrary number of sockets. Currently, this is implemented through multiplexing. As a result, this layer actually controls the whole processing flow. The multiplexing algorithm used for this purpose is depicted on Figure 3.

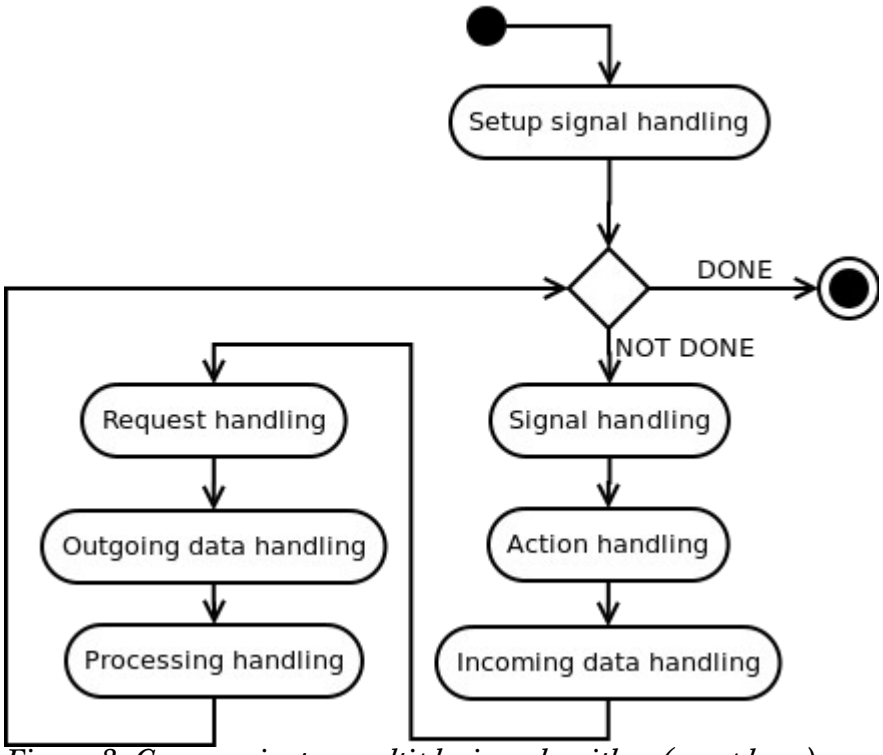

Figure 3: Communicator multiplexing algorithm (run phase)

One processing cycle consists of six phases. First all signals are handled. Through the signals, users may perform various tasks such as terminating the processing and exiting or dumping the processing statistics. Then scheduled actions are handled. The c*ommunicator* has a built-in scheduling mechanism which allows specific actions to be performed at a predefined time. Currently, this feature is used to implement socket reconnection with a back-off interval but generally any action can be scheduled at any application level. In the next phase, all incoming data is received on all sockets and requests are parsed out of the data string using the service of the upper *protocol* layer. Then the *protocol* layer is asked to attend to the awaiting requests. Subsequently all awaiting data is sent out from all sockets and finally the control is passed to the *engine* layer for short period of time so that the application can perform various processing, such as to read a few more lines from the watched log file, to pick up a few more messages contained in files, etc.

The next layer above the *communicator* is the *protocol* layer. On this layer the format of *protocol* messages and responses is defined; encapsulation and decapsulation is resolved and message delivery reliability is ensured. This layer provides a simple interface to *engines* for easy message transmissions. The data model for storing messages is not forced by this layer in any way so it can be used to transfer messages stored in any arbitrary format. There are no sockets on this layer, we refer to endpoints instead. This abstraction allows naming the peer side, socket reopening feature, etc.

The engine layer is hierarchically the highest layer. It contains the main processing logic. The *protocol* layer passes the control over to the *engine* to handle specific requests. For simple *engines*, the processing logic can be implemented directly within the *engine* module,

but for more complex processing or more reusable code it is much better to use the *cog* mechanism (see the Figure 4). *Cogs* are the building blocks of the *engines*. Each *cog* should perform a brief and strictly defined action on a given request, for example validation, event anonymization or normalization, saving into the database, etc. These *cogs* can then be chained together and configured to handle a particular type of requests within the *engine*. A failure in processing within one *cog* breaks the processing chain and an error is reported back.

Figure 4 displays the ideas mentioned in previous paragraphs. It also shows the data and request flow within all three layers. The final elements on the figure without any explanation are request queues and protocol level handled requests.

Queues are the mechanisms ensuring a reliable message delivery on the application level. There are two queues for each direction. Upon creation/reception, the new request is inserted into the *Incoming queue*. It stays there until it is processed. If the processing requires the new request to be sent to another destination, the current request is put on hold into the *Incoming pending queue* until the response for the next request is received or timeout elapses. If the response is received, it is forwarded as a response for the original request. If the timeout elapses, it depends whether the component is the original message source or has the message stored in some persistent storage, or whether the component relies on the original message source to resend it in case of failure. For performance reasons, most of the components store the messages only within the process memory, so all messages would be lost in case of failure. It is possible to set the component (e.g. the *concentrator*) to store messages in some sort of a persistent storage and make the transport chain shorter. However by default all middle components (*sensor*, *concentrator*) do not do that and rely on the message source to resend it if necessary.

Protocol level handled requests are requests that are not supposed to be handled on the *engine* level. The Transport protocol chapter describes the transport protocol and all defined requests and responses. Some of the requests are service and are not intended to be handled in a custom manner, for example *hello* (see the chapter Transport protocol for more details). These requests are then handled on the protocol layer and change the inner protocol structures.

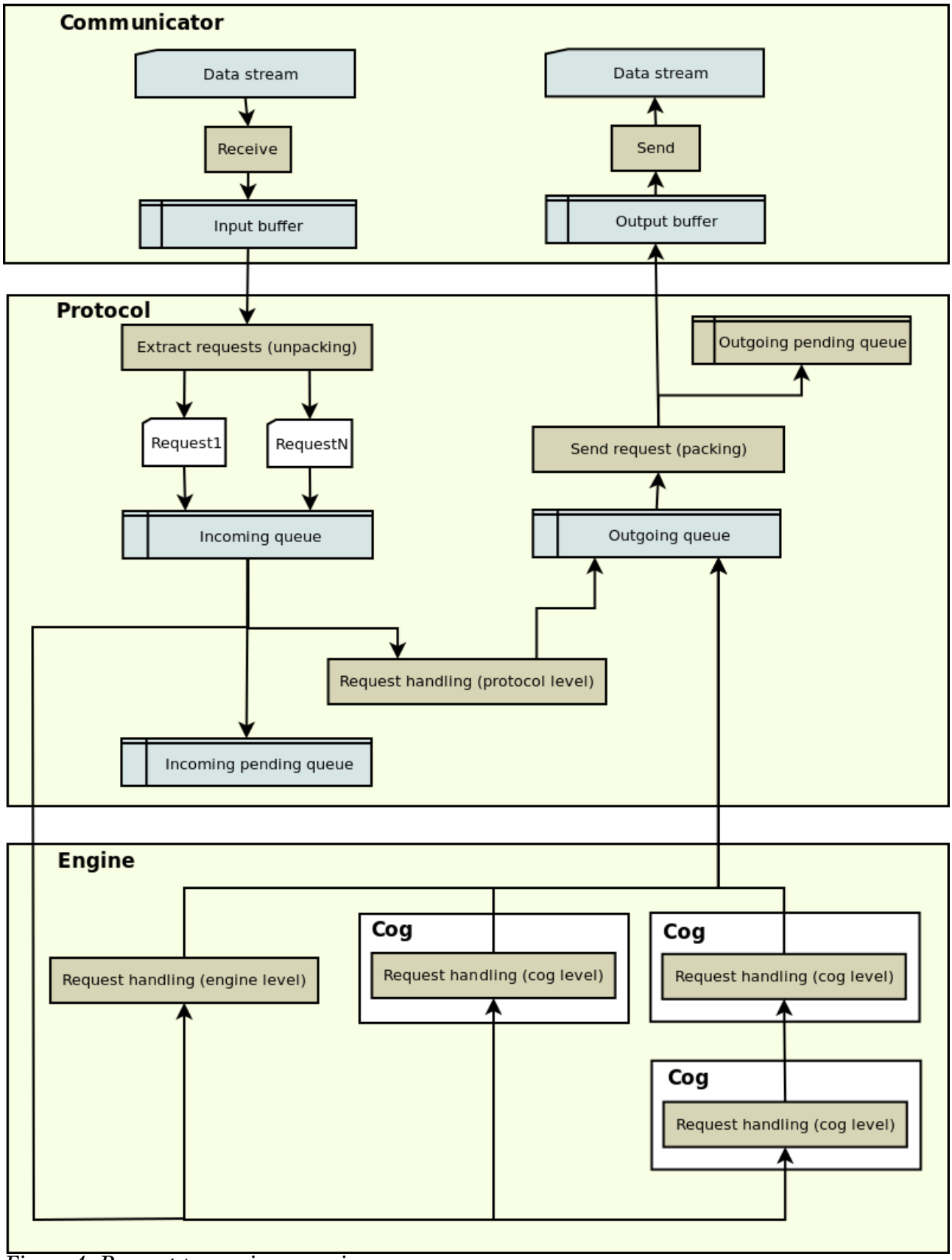

Figure 4: Request processing overview

Transport protocol

The transport protocol was inspired by the RELP [9] protocol which is used to ensure a reliable message delivery in the rsyslog [10] project. The main reason for implementing the custom protocol to ensure a reliable message delivery is the need to confirm the success after the the message on application (*engine*) level has been processed. We need to make sure that the message is safely stored in the database. Only then we can assume that it will not be lost.

It is a very simple command-response protocol with a defined set of commands. Each command has its own type of response. Although the set of commands is strictly defined, extending the protocol in the future with more commands is very simple. It is a bidirectional protocol; each request (command or response) is identified by unique ID, which is different for each side of the communication.

Every command has the following syntax:

```
SEQ SP COMMAND SP LENGTH [SP DATA] NL
```

Every response has the following syntax:

```
SEQ SP RESP SP LENGTH SP CMD_SEQ SP CODE [SP ADDITIONAL_DATA]
```

Elements in the request have the following meaning:

1. SEQ – integer, unique request ID, starting from 1 for first request and sequential, different for each side of the communication
2. SP – space (ASCII 0x20)
3. COMMAND – string, name of the command
4. LENGTH – integer, length of the request payload in bytes (excluding the first space)
5. DATA – command data
6. CMD_SEQ – integer, identifier of the previous command to which this response belongs
7. CODE – integer, response status code
8. ADDITIONAL_DATA – additional response data

The existing defined status codes are listed in Table 1. The status code table was inspired by HTTP status codes. 2xx numbers mean success, 4xx numbers mean error on your side and 5xx numbers mean error is on my side ("my" and "your" is meant from the response sender point of view).

There are only 2 commands and 2 responses defined by the current protocol version:

1. hello → resp-hello
2. alert → resp-alert

When the connection between two components is opened, the client (the one initiating the connection) must initiate a handshake and introduce itself. This handshake is done via

hello → *resp-hello* request pair:

```
SEQ SP hello SP LENGTH SP VERSION SP ENDPOINT_TYPE SP ENDPOINT_ID SP ENDPOINT_HOST NL
```

The response has the following syntax:

```
SEQ SP resp-hello SP LENGTH SP CMD_SEQ SP CODE SP VERSION SP ENDPOINT_TYPE SP ENDPOINT_ID
SP ENDPOINT_HOST NL
```

The new elements in the request have the following meaning:

1. VERSION – integer, protocol version
2. ENDPOINT_TYPE – enum, type of the endpoint (component): source, sensor, concentrator, drudger
3. ENDPOINT_ID – string, unique identifier of the endpoint (component) within the whole detection environment (setup), assigned by administrator
4. ENDPOINT_HOST – string, hostname of the host device running the current endpoint

After the handshake, it is possible to use the endpoint based addressing for request delivery. The protocol layer on each side of the communication pair can now perform additional validation so that inappropriate requests would never go to the component that is not capable of handling it.

The most important request pair is *alert* → *resp-alert*. It is used to exchange the alerts (messages) between the components. The command has the following request syntax:

```
SEQ SP alert SP LENGTH SP ALERT_CLASS SP ALERT NL
```

The response has the following syntax:

```
SEQ SP resp-hello SP LENGTH SP CMD_SEQ SP CODE NL
```

The new elements in the request have the following meaning:

1. ALERT_CLASS – string, name of the alert class
2. ALERT – serialized alert

Thanks to the *alert_class* element in the *alert* request,the *protocol* does not care how the messages have actually been serialized on the wire. Element *alert* can contain any payload and it is up to the higher application logic to deal with proper deserialization and handling.

Code	Meaning	Description (where needed)
200	OK	Success
201	Created	New entry/object was created in database/file system
202	Accepted	Command accepted
400	Bad Request	
401	Unauthorized	
403	Forbidden	
404	Not Found	Entry/object was not found in the storage
405	Method Not Allowed	
408	Request Timeout	
409	Conflict	
410	Gone	Entry/object does not exist in the storage
429	Too Many Requests	
499	Database Entry Already Exists	Entry/object already exists in the storage
500	Internal Server Error	
501	Not Implemented	
503	Service Unavailable	
507	Insufficient Storage	
597	Temporary Message Storing Error	Storage may be able to store the message later
598	Permanent Message Storing Error	Storage will never be able to store the message
599	Different Status Messages	Successive components returned different status codes

Table 1: Response status codes

Data model

Currently, we are using the Intrusion Detection Message Exchange Format (IDMEF) [11] as the underlying data model for event description and encapsulation. As stated before, majority of the system components and modules do not enforce this and are designed to be able to handle messages with any data model.

The IDMEF data model is defined in RFC 4765 and it has the XML data format. Initially, we chose it for many reasons For the sake of completeness, we provide an overview of advantages and disadvantages as we see them:

1. Advantages

1. XML – well defined, standardized, many existing tools (parsers, validators), many existing technologies (XSLT, XPath, XQuery), platform independent
2. RFC – someone had given it some considerations over a significant period of time
3. Designed specifically for exchanging messages between automatic systems
4. Very comprehensive, can store a wide range of heterogeneous information
5. Implemented by successful projects (Prelude IDS, Snort, Samhain)
6. Historical and compatibility reasons – initially we were testing Prelude IDS and maintaining the same data model was a step in the migration process

2. Disadvantages
   1. XML – too chatty, too complex
   2. It is aging, some elements and attributes are obsolete or unnecessary
   3. Structure is too deep – filtering rules are very long and ugly
   4. Some of the elements allow recursion, which is very hard for processing
   5. Does not directly support references, so it is not easy to add arbitrary data to specific elements
   6. Sometimes inconsistent
   7. Redundancy of some attributes (e.g. timestamps) is causing problems with priority

The list of disadvantages is quite long and some of them are causing us a lot of pain. Thus, in the future we are likely going to abandon the IDMEF and develop our own data format (based on the IDMEF).

We had to deal with the issue of addressing the nodes in the IDMEF XML DOM structure. A natural way of addressing a particular node in XML is XPath [12]. But from our point of view XPath had the following disadvantages:

1. Too powerful to be given to the user (one that would be configuring the filtering rules etc.)
2. Addressing rules would be too long because of the need to specify the message root element and namespace name for each node

We got the idea from Prelude IDS and created MPath language. MPath is a subset of XPath and internally it is translated to it, before the rules are applied to the message. Syntax must follow these simple rules:

1. The '/' character is used as a node separator
2. Attribute names must begin with '@' character
3. After the element name, it is possible to use index enclosed in square brackets '[' and ']', but the index can only be an integer

This MPath is then used to create all all messages and to perform all manipulation tasks

and database queries. We use a very simple key → value mechanism to create a message and to map real events to the IDMEF data model. The key is the MPath address of the designated element. For example:

```
Alert/Analyzer/@name                       => LaBrea
Alert/Analyzer/@class                      => honeypot
Alert/Source/Node/Address/@category        => ipv4-addr
Alert/Source/Node/Address/address          => $1
Alert/Source/Service/@iana_protocol_name   => tcp
Alert/Source/Service/@iana_protocol_number => 6
Alert/Source/Service/port                  => $2
Alert/Target/Node/Address/@category        => ipv4-addr
Alert/Target/Node/Address/address          => $3
Alert/Target/Service/@iana_protocol_name   => tcp
Alert/Target/Service/@iana_protocol_number => 6
Alert/Target/Service/port                  => $4
Alert/Classification/@text                 => Connection attempt
Alert/Assessment/Impact/@severity          => medium
Alert/Assessment/Impact/@completion        => failed
Alert/Assessment/Impact/@type              => recon
Alert/Assessment/Impact                    => Remote host $1 connected to honeypot
Alert/Assessment/Action/@category          => block-installed
Alert/Assessment/Action                    => Connection tarpitted
Alert/AdditionalData[1]/@type              => integer
Alert/AdditionalData[1]/@meaning           => cease-time
Alert/AdditionalData[1]/integer            => $5
Alert/AdditionalData[2]/@type              => integer
Alert/AdditionalData[2]/@meaning           => count
Alert/AdditionalData[2]/integer            => $6
Alert/AdditionalData[3]/@type              => string
Alert/AdditionalData[3]/@meaning           => log_line
Alert/AdditionalData[3]/string             => $7
```

In the previous code snippet, we have created an IDMEF message describing the connection attempt to the LaBrea honeypot. Note the '$X' on the right side of the rules. These are variables and are substituted on the fly with real values. The previous code snippet is the template according to which it is possible to create multiple messages. The example also demonstrates the index MPath feature. Note the *AdditionalData* elements at the end of the rule list.

Persistent data storage

The requirements posed on persistent data storage are very high. It must be scalable, very fast and easy to manage. We want to be able to store as much data as possible, run non trivial searches through them and get the results in a short time (imagine a member of a CSIRT team solving an incident regarding some IP address – how long is he willing to wait?).

IDMEF is a XML document, so after some experimenting with RDBMS we chose MongoDB [13] which is a very fast, horizontally scalable document oriented NoSQL database. NoSQL databases do not store data in table and row and not require fixed table schemas as in RDBMS. MongoDB stores data in collections equivalent to tables in a RDBMS; and documents equivalent to rows in a RDBMS. It stores them in the BSON (Binary JavaScript Object Notation) format identical (semantically, it is a binary serialization) to standard JavaScript Object Notation (JSON) for most structures. BSON also contains ex-

tensions that allow representation of data types that are not included in the JSON specification. BSON, for example, has both *Date* and *BinData* data types.

MongoDB's schema-less design allows creating documents with a variable structure which is exactly what we need. Obviously, IDMEF message describing a SSH port scan will be very different from the message describing a phishing attack. Storing such variable data is trivial in MongoDB.

Hawat GUI

The Hawat graphical user interface mentioned at the beginning of this report is currently under heavy development. It was mentioned only to complete the picture of the system and it will be described in detail in a future report.

Conclusions

Mentat is our attempt to consolidate gathering, storage and processing of intrusion alerts we receive both from our own detection systems and honeypots and from some third party services such as Shadowserver and Team Cymru. It is a distributed system capable of detecting security events, or just receiving them from third party software or service, transporting them securely through the network of components and delivering them to a searchable persistent storage and to other processing modules. Currently we have a prototype, a proof of concept which is running in a test mode with a couple of message sources attached to it. The system is under rapid development. Based on the testing we amend the design or implementation mistakes, add new features and secure and optimize the whole system.

At present we consider the Mentat as a framework or a platform for ideas. We will see where this path will take us in the future. It is a platform for testing detection mechanisms; a new detection module can be implemented very easily by extending and implementing prepared interfaces. We aim at intelligent anomaly detection based on network traffic monitoring and detecting weird or suddenly different activity. In the future, we would like to make this project available to the community.

Acknowledgements

This work is supported by the research intent MSM6383917201 of the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1]: Shadowserver Foundation, http://www.shadowserver.org/wiki/

[2]: Team Cymru, http://www.team-cymru.org/

[3]: OTRS, http://www.otrs.com/en/

[4]: Prelude IDS, http://www.prelude-ids.com/en/

[5]: OSSEC, http://www.ossec.net/

[6]: OSSIM, http://communities.alienvault.com/
[7]: Postfix Documentation: Architecture, http://www.postfix.org/OVERVIEW.html
[8]: Catalyst MVC Framework, http://www.catalystframework.org/
[9]: RELP, http://www.librelp.com/relp.html
[10]: rsyslog, http://www.rsyslog.com/
[11]: RFC 4765: IDMEF, http://www.ietf.org/rfc/rfc4765.txt
[12]: XPath, http://www.w3.org/TR/xpath20/
[13]: MongoDB, http://www.mongodb.org/