Warden 3: Internet Threat Sharing Platform

Pavel Kácha, Michal Kostěnec, Andrea Kropáčová

***Abstract*—**For large existing body of automatically detected security events, be it honeypot machines or IDS systems, golden mine of netflow data or log data of production machines, manual distribution is infeasible. The Warden project is a platform for automated sharing detected security events among security teams. Involved parties can expand their own detected threat stream by events from other members, and vastly improve their security threat evasion and knowledge about network health. Clients, connected to Warden, can use incoming data as early warning systems, data mining and analysis engines, reputation databases, blacklist or firewall rule generators or just a data storage pools for history and trend analysis. This paper describes the design and implementation of Warden 3, the fundamental rewrite of previous version, taking advantage of nowadays technologies, using flexible JSON based Intrusion Detection Extensible Alert (IDEA) format, and aiming for robustness and solid performance.

***Keywords*—**alert, security event, incident response, ids, event exchange, honeypot, json

I. Introduction

Warden is a system for efficient sharing information about detected threats, available under 3-clause BSD license. The system mimics the behaviour of the queue with multiple producers and multiple consumers – the detection probes push security events to the hub, and clients – analysers, blacklist generators, storage and aggregators can pull the new events at will. However, to take some burden out of clients and network, the server provides means for basic filtering.

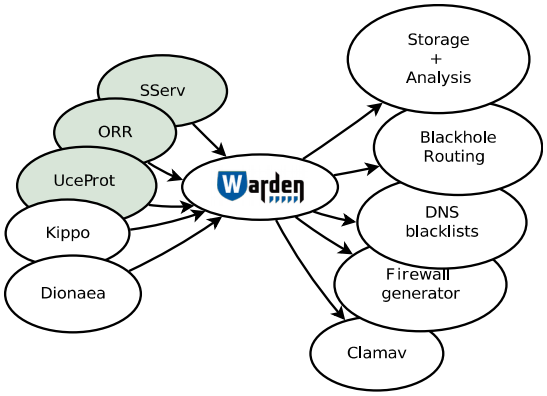

Fig. 1: Warden architecture

The work has been supported by the CESNET association and the operator of the Czech National Research and Education Network referred to as CESNET2 within its "Large Infrastructure" (LM2010005) research programme, running within 2010–2015 timeframe.

Pavel Kácha, Michal Kostěnec and Andrea Kropáčová work in CESNET,, a. o. l. e., Zikova 4, Prague, Czech Republic (e-mails: ph@cesnet.cz, kostenec@cesnet.cz, andrea@cesnet.cz).

It is safe to say that venerable Warden 2 [16] was ambitious project and vast improvement in security incident sharing in CESNET network – the identification of new requirements and need for new directions is itself a proof that it is useful and project itself was a great achievement.

Warden improved security incident handling speed in CESNET NREN and helped to reach healthier network state. Its vast body of incident data – be it from internal detectors, or from third party sources [15] – allowed for interesting data processing and correlation projects, such as [1] and [9].

II. Requirements

However, based on several years of life with previous version of Warden, real world experience and also taking into account current state of the art, let us identify possible improvements, suitable changes, and new requirements.

A. Previous Warden version review

Warden 2 used RPC calls for sending and receiving events using SOAP over HTTPS. While HTTPS shows its utility, SOAP with its complete XML (de)marshalling stack bites into performance and brings in large tree of library dependences. There is a need for lighter and more maintainable approach.

Events were represented as number of RPC call arguments (name, time, type, attack source IP and type, attack destination port, attack volume and free text note). That has shown as insufficient for many of security events nowadays in the wild (complex phishing attacks as a notable example). We need to find more flexible and extensible representation.

Also, code for validation of events must have been written by hand, specifically for our defined fields. Some standard solution would be more robust.

Warden 2 server provided basic event filtering for receiving clients, based on event type, and on simplistic notion of "own" events. This proved insufficient, users are calling for filtering based on detector type (honeypots provide greater certainty than portscan detectors, for example). Also, "own" means something different for various users, especially in organisations with complicated internal hierarchy – we need to use better representation.

Event type and detector description type was represented by loose set of categories (tags), which were added on "as needed" basis. We need to use some more standard and structured solution.

Sending API was designed to push only one event at a time,

severely limiting throughput and overall sender performance.

Clients needed to save state – while this is good feature to have for complex receiving clients workflow, there is no need for simple pull-process-forget clients to be stateful.

In Warden 2, 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. We should introduce more tamper proof solution.

Last but not least – Warden 2 server was written in Perl. While it was logical choice when the project started, nowadays, when Perl 3 (and necessary ported libraries) is nowhere near to finish, number of skilled Perl programmers is on the decline, and stability and quality and compatibility of requisite libraries for Warden 2 varies wildly, even language and platform is worth reconsideration.

III. Design

Lets now elaborate on stated requirements and make implementation decisions.

A. Protocols

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 XML (which is driving engine SOAP in previous Warden version), 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.

B. Formats

The serialisation protocol closely relates to format of security events. We have already (originally for Mentat project [9]) created structured and extensible format – IDEA [5], which already uses JSON as main representation.

With IDEA we also get mature incident categorization (based on MkII [14]) and expressive set of detector description tags for free [6].

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

C. 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).

D. 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 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.

E. 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.

F. 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.

G. 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.

H. 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 MTA. Each mail message, entering the system, receives sufficiently unique identifier, which gets propagated to all log messages, appears in 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.

I. Platform

Experience shows that Perl is not ideal choice anymore, however we would like to stay with flexibility, rapid prototyping and deployment speed of dynamic scripting language. We need solid library support for JSON and HTTPS on client side and support for high performance data based (non HTML) web application, along with good database support.

As Warden client library is only a communication channel, on which other third party data processing applications will be based, we also have to consider user base scope, libraries and frameworks support.

Our choice fell naturally to Python, based also on experiences on other projects. Python standard library already provides HTTP and HTTPS support, work with X509 certificates for authentication, WSGI support for connectors to powerful web server software, solid database support, and also handful of scientific frameworks (namely NumPy [11], SciPy [13] and Matplotlib [10]).

We also need the high-performance communication channel. Python brings in WSGI [4], the interface API web servers. There exist WSGI connectors to various web server software, among others to venerable Apache HTTP server, to which we can trustfully offload the performance burden of HTTP(S) implementation.

IV. HTTP API DESIGN

Leaving SOAP creates possibilities for arguments and results representation. We 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 [3]*, 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.

A. 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
  - *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.

B. 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

C. getEvents method

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

1) 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)

2) Returns

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

3) 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"}]}
```

D. sendEvents method

Uploads events to server.

1) Arguments

- POST data – JSON array of Idea events

2) Returns

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

3) 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}
```

4) 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
}
```

E. getInfo method

Provides client with basic server information.

1) 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

2) 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"}
```

V. Database design

A. Essential queries

Lets take a look at the queries, which will create focus of the server work.

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.

B. Discussion

It is clear that we need the table of events and table of clients and positive relation between them.

Concerning events table, the only information we need to parse out from arriving JSON events are the filtering fields, so we do not need to try to represent the whole IDEA structure in database. However, this is in fact mainly Category array, which means we have one to many relation and we will have to split these into separate table. The same applies to the detector tags.

Both tags and categories are transferred as free text strings, which shows as a performance and space intensive way to represent them in relations. We have also tried conversion to hashed fixed length strings, which lessened impact, however we have still felt, that there is a margin. However database itself does not need to work with text identifiers directly in the queries, so we will create mapping of these finite sets to integer sequence and use these quite short integers in the database representation.

Interesting situation arises in connection with detector names – we have to be able to filter by prefix substring – *"org"*, *"org.example"*, *"org.example.honeypot"* can all be used as patterns. One possibility is to create auxiliary table with all possible prefixes, however that shows very bad performance in negative queries, where database is forced to generate large list of all non matching prefixes on which it consequently filters event data. However database indices are indeed prefix based, so correctly used anchored *LIKE* operator is enough.

Next complication we have to solve is saving last pull ids of accessing clients – straightforward solution would be to store it in the table of clients. However, table of clients is mostly immutable, and from administration point of view it would be wise to leave access to it only to human operator, avoiding frequent changes by server itself, we will thus split this information into last_events table in the form of client identifier, last event id and login timestamp.

Concerning database engine itself – we prefer raw performance over capabilities – we can miss a few security events in case of outage if server is able to withstand large number of concurrently accessing client connections. Our first choice points to MySQL, whose InnoDB engine supports reasonable number of database capabilities and fine grained line based locking (necessary for concurrent read/write access) together with decent performance.

C. Final schema

```
CREATE TABLE events (
 id int(11) NOT NULL AUTO_INCREMENT,
 received timestamp NOT NULL,
 client_id int(11) NOT NULL,
 `data` longtext NOT NULL,
 valid tinyint(1) NOT NULL DEFAULT 1,
 PRIMARY KEY (id),
 KEY id (id, client_id)
);

CREATE TABLE event_category_mapping (
 event_id int(11) NOT NULL,
 category_id int(11) NOT NULL,
 KEY event_id_2 (event_id, category_id)
);

CREATE TABLE event_tag_mapping (
 event_id int(11) NOT NULL,
 tag_id int(11) NOT NULL,
 KEY event_id_2 (event_id, tag_id)
);

CREATE TABLE clients (
 id int(11) NOT NULL AUTO_INCREMENT,
 registered timestamp NOT NULL,
 requestor varchar(256) NOT NULL,
 hostname varchar(256) NOT NULL,
 note text NULL,
 valid tinyint(1) NOT NULL DEFAULT 1,
 name varchar(64) NOT NULL,
 secret varchar(16) NULL,
 `read` tinyint(1) NOT NULL DEFAULT 1,
 `write` tinyint(1) NOT NULL DEFAULT 0,
 test int(11) NOT NULL DEFAULT 0,
 PRIMARY KEY (id)
);

CREATE TABLE last_events (
 id int(11) NOT NULL AUTO_INCREMENT,
 client_id int(11) NOT NULL,
 event_id int(11) NOT NULL,
 `timestamp` timestamp NOT NULL,
 PRIMARY KEY (id),
 KEY client_id (client_id, event_id)
);
```

VI. 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.

A. 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)
```

1) 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

2) Returns

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

B. 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.

1) Arguments

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

2) Returns

Dict, prepared from read JSON data.

3) Example

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

C. 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.

1) 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

2) Returns

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

D. warden.Client.sendEvents

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

1) 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

2) Returns

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

3) 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). See also IV.A Error handling.

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.

E. warden.Client.getInfo

```
wclient.getInfo()
```

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

F. 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.

VII. Performance

Nowadays there is 36 millions of events in the Warden database of average event size 786.3 B. Clients produce on average 7 events per connection, however bunches of 1000 events (selected as reasonable maximum) are not rare. However median is 2 events per connection, so we will have to make sure final Warden 3 performs well both on many small accesses and on bulk uploads.

Indicative testing of prototype, based on this design, shows that practical limit may be somewhere about 40 000 incoming events per second on single event per connections, however throughput seems to be able to raise at least up to 110 000 events per seconds when clients use 100 events per connection – this indicative test was made using up to 80 simultaneously accessing clients and shows that outlined design is worth pursuing.

VIII. Conclusion

Warden 3 is complete redesign, based on the identified shortcomings emerged during several years of Warden 2.X operation. Which is not to lessen merit of Warden 2, it is necessary to note that without it, Warden 3 wouldn't most probably exist.

New Warden uses flexible and descriptive event format, based on JSON. Warden 3 protocol is based on plain HTTPS queries with help of JSON (Warden 2 SOAP is heavyweight, outdated and draws in many dependencies). Clients can be multilanguage, unlike SOAP/HTTPS, 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, and uses MySQL as efficient data storage. The Python WSGI layer is run under venerable Apache web server.

The performance characteristics show, that despite providing more features and working with much more complex event format, Warden 3 outperforms previous version in orders of magnitude, and should be able to withstand very significant peak loads.

IX. Future work

Some of the Warden 2 clients were already converted to prototype Warden 3 instance with superb results, however turning the prototype into full production state along with subsequent transferring of existing clients will still need a lot of work.

However, family of tools, based on Warden client library, will be able to emerge, namely connectors for Kippo and Dionaea honeypots, connectors to storage and data mining tools and many others – able to work with much wider pool of information, accessible in Warden's new extensive security event format.

X. Acknowledgment

The access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum (http://www.metacentrum.cz/en/), provided under aforementioned programme, is highly appreciated.

References

[1] V. Bartoš, M. Žádník, *An Analysis of Correlations of Intrusion Alerts in an NREN*. 19th International Workshop on Computer-Aided Modeling Analysis and Design of Communication Links and Networks (CAMAD). IEEE, Athens, December 2014. ISBN: 978-1-4799-7134-3.

[2] D. J. Bernstein, *Using Maildir format* (original specification), Cited 3 Sep 2015. Available: http://cr.yp.to/proto/maildir.html

[3] *cURL*. Cited 29 May 2015. Available: http://curl.haxx.se/

[4] P. J. Eby, *Python Web Server Gateway Interface v1.0.1*, 26 Sep 2010. Available: https://www.python.org/dev/peps/pep-3333/

[5] P. Kácha, IDEA: Designing the Data Model for Security Event Exchange, *17th International Conference on Computers: Recent Advances in Computer Science*, Rhodos, 16 July 2013, ISBN: 978-960-474-311-7, ISSN: 1790-5109.

[6] P. Kácha, IDEA: Security Event Taxonomy Mapping, *18th International Conference on Circuits, Systems, Communications and Computers: Advanced Information Science and Applications*, Santorini, 17 July 2014, ISBN: 978-1-61804-236-1, ISSN: 1790-5109.

[7] P. Kácha, *Incident Classification Comparison (with eCSIRT.net mkII as main reference)*, CESNET, 10 January 2014. Available: https://csirt.cesnet.cz/_media/en/idea/incident_classification_comparison.ods

[8] *Postfix*. Cited 23 Sep 2015. Available: http://www.postfix.org/

[9] J. Mach, *Expert system Mentat*, CESNET 9 December 2013. Available: http://www.cesnet.cz/wp-content/uploads/2013/12/mentat-paper.pdf

[10] *matplotlib*. Cited 29 May 2015. Available: http://matplotlib.org/
[11] *NumPy*. Cited 29 May 2015. Available: http://www.numpy.org/
[12] J. Safarik, M. Voznak, J. Slachta, L. Macura, F. Rezac and J. Rozhon, Modular system for gathering and classification of SIP attacks, *19th International Conference on Circuits, Systems, Communications and Computers 2015: Recent Advances in Computer Science*, Zakynthos, July 2015. ISBN: 978-1-61804-320-7.
[13] *SciPy*. Cited 29 May 2015. Available: http://www.scipy.org/
[14] D. Stikvoort, *Incident Classification*. 23 May 2013. Available: http://www.terena.org/activities/tf-csirt/meeting39/20130523-DV1.pdf
[15] P. Vachek, CESNET CSU System, *16th WSEAS International Conference on Computers 2012: Recent Researches in Communications and Computers*, Kos Island, July 2012, ISBN: 978-1-61804-109-8.
[16] *Warden*. CESNET. Copyright 2010-1013. Last updated 17 April 2013. Available: https://wardenw.cesnet.cz/en/index

Pavel Kácha (CESNET) has worked in computer security, software development and system administration over 12 years. He is a member of the CSIRT team of the CESNET association where he is responsible for several security incident handling related projects. He also participated in establishing CSIRT.CZ – the Czech national CSIRT team. He specialises in timely security incident detection and information dissemination and enhancing a general security awareness.

Michal Kostěnec (CESNET) focuses mainly on computer networks and security. Since 2009 he is a network specialist at the University of West Bohemia, where he also participates on increasing of MAN WEBnet security and develops network oriented tools. Nowadays he is a member of the CESNET-CERTS security team, where he is responsible for honeypot administration, development and distributed software testing. He is also a member of the Forensic Laboratory team (FLAB), where he conducts penetration and performance tests.

Andrea Kropáčová (CESNET) specialises in computer network security and security of services operated on the network, with the emphasis on the handling of security incidents, their detection and prevention. She is the head of the CESNET-CERTS security team, the first officially established CSIRT-type team in the Czech Republic (in 2004), operating above the CESNET2 network. In the framework of the grant of the Czech Ministry of Interior, she and the CESNET-CERTS members established the CSIRT.CZ team, which currently serves at the Czech National CSIRT.