Crunching Logs for Fun and Profit

Radoslav Bodó, Daniel Kouřil, Jiří Sitera, Miloš Mulač, Pavel Vondruška, Michal Procházka (?)

Abstract

Contents

1 Introduction

Centralized management of logs produced by computer systems is an important piece of operations. Having log records collected at a single point simplifyies handling of various incidents since the operator has all the information collected centrally. For example a common task is to detect what steps a particular has performed recently, which is needed during an investigation of a security incident. Without a centralized log repository it is necessary to connect to every single machine in the infrastructure and check all the logs that may relevant. The whole process is too long and one risks that an important piece of information is missed. On the other hand, collecting such data collected on a single place can be done very quickly. Another example is ongoing checks of the logs, e.g. to detect an issue immediately when it appears. Again, without centralized log management in place, all relevant machines in the infrastructure need to perform these checks independtly, which is hard to establish and maintain. Being able to evaluate logs quickly and work with them on the fly contributes to sound system management and makes it more efficient.

Despite the basics of such a deployment is well understood, we identified several areas that either are not covered at all or whose scope exceeds the possibilities of current tools.

In this report we describe the deployment of a central logging service in the Czech NGI and means how the data collected is processed. There are basically two crucial areas that need attention, which have been identified during the deployment. First off, it is necessary to obtain logs produced by all the nodes in the infrastructure. This is easy to be done in a single institution for which common tools available today are just sufficient. However, if logs are collected from multiple different institutions and/or the infrastructure gets more complex, issues appear that did not expose at the smaller scope or were even not deemed important. For instance, while collecting logs from a local environment it is usually acceptable for the institution to rely on physical security of the local network, which also provides a reasonably stable environment, so log records sent over UDP get lost only rarely. On the other hand, log collectors receiving data from multiple institutions (or multiple branches of a single institution) that are interconnected via the Internal must ensure confidentiality of the data transferred as well as a high level of fault tolerance. In the paper we describe our deployment emphasizing the changes that had to be done in order to achieve a secure and reliable passing of logs.

Another principal problem concerns with the way how data is processed once it is collected and stored on the central service. The traditional way is setting up series of filters that detect some known patterns in incoming data. This mechanism can only be employed for patterns that are known in advance and can only be applied to new data. Sometimes it is therefore necessary to analyze the data collected. There are several common tools that can be utilized for that, like grep and other standard Unix commands. However, processing a large amount of data with these tools is very intensive, which makes them unusable for interactive work.

Based on these limitations we decided to explore alternative ways how logged data can be processed and examined. We combined together several open source solutions and built up an infrastructure to fast index log record and manipulate them in an easy and quick way. The solution is based on an internal cloud that runs the indexing tools, which continually process received logs. Being provided in a cloud, the indexing service can shrink or enlarge on demand, based on current amount of the data and requirements of the operators. On the top of the pre-processed results we run visualization tools that are used to access the data and manipulate with it via web interfaces.

Having a centralized service collecting data from the whole NGI and tools to efficient manipulation with the data, made it possible for us to have a better control over the infrastructure. The fast access to information collected is appreciated by operators, infrastructure developers and also by the security officers. The possibility to interactively work with all logs also opened new possibilities of how this data can be utilized. For instance, it is much easier to detect misconfigured services or uncommon events, which would cause serious problems if they were not handled.

The rest of the report is organized as follows ....

2 Collecting Logs

Applications generate log records describing important events in their processing. The records have to be stored in a permanent storage for further use. In order to log an event the application use either an own mechanisms to store it in a file or database or use a dedicate library. For the latter case the syslog POSIX calls are often utilized, which usualy pass the log records to the syslog daemon running on the system. In the default configuration the syslog deamon stores log messages on a disk but it can also be configure to send them to a remote server over a network. In order to deliver log messages, the syslog messaging protocol[6] is used. If the logging is done over e.g. a private network and do not need any security, it is very easy to establish. Since the log messages may carry sensitive information, it is reasonable to ensure they are protected properly. To provide transport security of the log messages, the TLS protocol can be used[9], which is supported by the main open-source implementations. Remote logging, including its protection using TLS is properly documentued and sources exist that describe it in details (e.g. [10]).

However, we were deploying the solution in the environment of project MetaCentrum[**?**], which relies heavily on the Kerberos security system. Deploying a paralell PKI infrastructure would have been an overkill, so we reqired to secure the messaging using Kerberos, too. Another requirement stem from the fact that the infrastructure is distributed all over the whole country, which increases chances of network outages. Therefore, a basic requirement for the whole messaging was that it provide sufficient fault-tolerancy so records do no get lost when they cannot delivered for a while.

Based on these requirements we selected `rsyslog` as the implementation of the syslog messaging and server. Daemons of rsyslog are standard part of major Linux distribution and most of them utilizes it as the default solution to store logs. It has a plugin-based architecture supporting various ways of authentication and channel protection. One of the plugins supports GSSAPI, which would allow us to leverage the existing security infrastructure of MetaCentrum. However, after we soon found out that the module from the stable relase of `rsyslog` (5.8 at the moment of writing) was crashing from time to time and initiated a fix, which was introduced in 5.8.x and onwards. However, even after the fix the plugin is not an ideal solution for a huge distributed environment. Having done quite extensive stress testing, we found out that the client side of the plugin is not able to reconnect correctly on server failures, which yields a huge traffic triggered by TCP reconnection attempts. Therefore, we have implemented a new GSSAPI module (`omgssapi`), which is based on the code of output plugins for the 5.8 family which merged the GSSAPI functionality from the existing module. The new module has been contributed to the community[1] and is available from *MetaCentrum REPO*[11].

2.1 Rsyslog shipper

It is easy to instruct the rsyslog deamon on a node to pass every logs it receives to a remote host. A simple setup needed is described in [7]. However, in order to provide a robust delivery, a few additional options are needed, for which it is very important to understand how the internal message processing works[2]. By default every rsyslog action invoked to deliver a message is processed directly in the context of Main rsyslog message queue and the corresponding thread. If the action is long running or is failing and retrying (possibly forever), it can block processing of the other messages. To address this issue, it's advised to use one of the available action queue implementations to de-couple the processing of the action to separate queue/thread. A general recommendation is to use a disk-assisted queue, which makes it possible for a queue to buffer its contents on a disk and to process actions asynchronously so the main processing is not blocked.

When using disk assisted queue implementation for tcp or gssapi (or other actions, the buffering can lead to filling whole disk space, which is not desired in production environment. In order to prevent this the queue must be restricted to use only defined maximum space. It is also inevitable to configure additional enquing timeout on the queue so it starts dropping messages that cannot

[1] `http://www.mail-archive.com/rsyslog@lists.adiscon.com/msg05902.html`

[2] Detailed information about queue management can be found at `http://www.rsyslog.com/doc/queues.html`

be insterted and protect rsyslog from blocking indefinitely[3].

Final configuration for rsyslogd shipper is as follows:

```
$ActionQueueType LinkedList          # use asynchronous processing
$ActionQueueFileName srvrfwd1        # set file name, also enables disk mode
$ActionResumeRetryCount -1           # infinite retries on insert failure
$ActionQueueSaveOnShutdown on        # save in-memory data if rsyslog shuts down
$ActionQueueMaxDiskSpace 100m        # limit disk cache
$ActionQueueTimeoutEnqueue 100       # dont block worker indefinitely when cache fills up
*.* :omgssapi:<server_name>:<port>   # deliver all messages to central server using GSS-API protection
```

Figure 1: rsyslog shipper configuration

Note that the Action Queue parameters are set separately for each action (sending to the remote in our case). So the setting in the snippet does not influence existing rules in the syslog configuration. It can be safely added to `/etc/rsyslog.conf.d/`.

2.2 Rsyslog server

TODO: pridavani autentizovane hostname?

The configuration of a server collecting logs is quite straightforward and is also well described in the rsyslog documentation. Since it is not possible to combine multiple authentication mechanisms in a single connection, the server needs to listen to multiple TCP ports and clients needs to connect to appopriate port to perform requested authentication.

Server side is being done by default GSS-API input module. A snippet of it's configuration, enabling GSS-API support is on fig. 2.

```
$ModLoad imgssapi
$InputGSSServerServiceName host
$InputGSSServerPermitPlainTCP off
$InputGSSServerRun 515
$InputGSSServerMaxSessions 2000
```

Figure 2: rsyslog server configuration for GSS-API

Logs received can be forwarded to another service for processing or stored on the system in file or databases. When it is stored, the backend must be choosen properly to allow easy processing of the data when needed. We are used to store the the data in a file hierarchy based on the IP addresses of the client.

The system must be properly monitored to make sure the syslog server accepts messages properly. It is important to note that while outages are handled gracefully by the clients, a longer outage would lead to dropping messages by the client, which opens a window of opportunity for attackers.

2.3 Virtual testbed

Despite that rsyslog internal architecture is well documented, development of multihreaded application working with strings and network is always an error prone proccess. Extensive testing is needed to ensure that installed version of important system component will not criple whole computing environment in a matter of seconds.

In the beging we used only two preallocated/reserved cluster nodes to develop an early version of the new GSS-API module. But using just two nodes to *fully* simulate various race condition situations was not really possible. Because of this, we decided to use a distributed testing environment based on MetaCentrum's virtualization framework[8, 4].

[3]Details on the setting can be found at `http://lists.adiscon.net/pipermail/rsyslog/2012-March/029580.html` and `http://www.mail-archive.com/rsyslog@lists.adiscon.com/msg08082.html`

Framework allows us to dynamically create a set of new virtual workernodes. Those are are instructed to install a new version of rsyslog and emit a certain ammount of messages toward a testing instance of central rsyslog server. During a test we also perform several (non) standard situations like restarting cental rsyslog daemon or killing tcp connections using tcpkill. Whole test is automated by set of scripts performing neccessary actions.

Virtual workernodes are created by framework using Magrathea bootstrapper (component responsible for instantiating a VM based on selected image) and creating of such nodes in the grid is scheduled by Torque batching system[5]. Torque interleaves *normal jobs* along with our *cluster jobs*, so we can use free cluster resources and not to interfere with standard workernode installations and not affecting other computing elements or standard user workernodes.

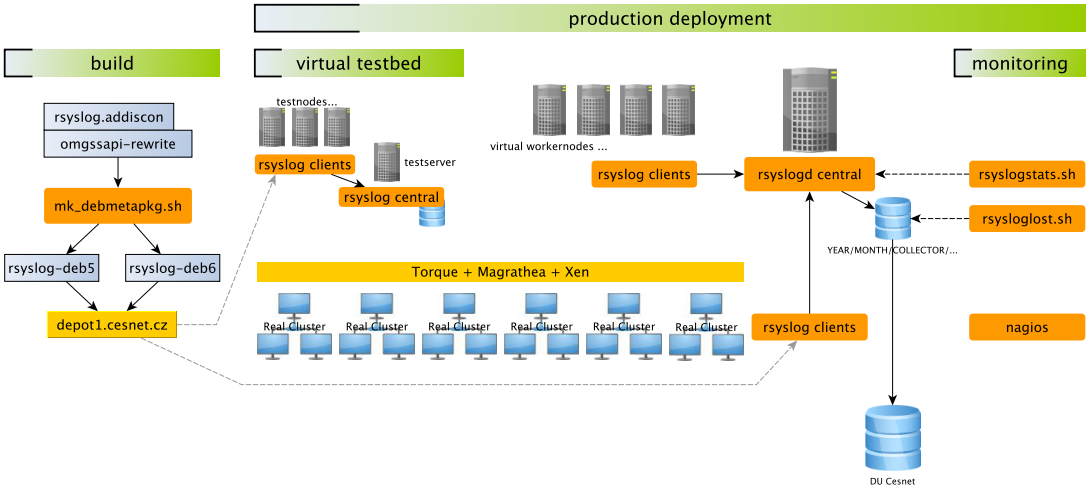

Figure 3: rsyslog build, test, deploy

3 Log processing

Once all system logs are stored on one place, it is possible to analyse them. A lot of various things can be monitored and extracted out of the log records. For example, the Torque batch system logs information about resource utilization (cpu, memory) of the jobs, which can be used to extract information about the actual utilization of the resources. This information can be communicated to the users whose jobs exceed granted capacities and possibly used to enforcement of the limits. The central storage makes it much easier to collect information about activities of a particular user in the whole environment. Being able to access this information on demand is crucial for efficient function of both user support and security operational staff.

The common way how logged data is processed nowadays is to use general text handling tools like `grep`, `awk` or `perl`. While this approach works for simple cases, the task is getting complicated by the ever increasing size of the logs and also by the fact that subsequent queries requires grepping the whole set over and over again. Additional analytic tasks require efficient aggregation queries, scanning through the whole dataset and also some persistent storagge for internal data and metadata.

The traditional solution to handle large volumes of data is to utilize relational databases. However, before taking the straightforward approach we decided to consider a few things:

The size In the grid environment it might not be sufficient to rely on single node database system because ammount of logs is waste. A scaling solution must be used by design. Probably advanced partitioning or sharding features of traditional RDBMS like MySQL or PostgreSQL can be used.

The structure In the traditional RDBMS it is necessary to know in advance and work with the fixed structure of the data stored in the tables. Every change of the structure must be reflected not just by application using that data, but also by the corresponding database model and configuration. This requirement does not quite fit the way how logs are processed since it is often not known in advance what kind of data will be processed. For instance logs generated by Apache differ a lot from logs produced by Kerberos KDC.

Those charateristics led us further into exploring new emerging technologies like text indexing and NoSQL databases, in particular we focused on ElasticSearch and MongoDB. Both solutions are designed to work in a cloud-like environment and both employ a structureless approach to work with the data – json documents. The first aspect is very likely to fit into grid computing environment possibly to address size and scaling issues. The second is likely to be more than helpfull for fast prototyping or research development of the applications working with logs.

3.1 ElasticSearch

ElasticSearch[4] is a full-text engine to search and index text data, which is built on the top of the Lucene library[3]. ElasticSearch was designed for a cloud, allowing for dynamic adding and removing of nodes, based on current requirements. It features several charateristics that makes it easy to use. In particular it supports autodetection of other ElasticSearch nodes on the same network, automatic splitting and distribution of the indices among the available nodes and their replicating and rellocating when needed. In order to search the indexed data users can utilize available interfaces (REST, native, thrift) or a web GUI provided by Kibana[5].

ElasticSearch can run as a single process but its advantage is its p2p character, which allows to dynamically add and remove its instances on multiple nodes within a local network. Therefore, it is possible to provide resources just according to current needs. More, every instance can play one or more roles of the node configuration (*client*, *master*, *data*) which is used for deploying a cluster with layered architecture. However, current interfaces do not support encryption or authentication and anyone who can talk to the cluster will be able to work witht the data stored in the index. To address this limitation it is advised to make sure the cluster of ElasticSearch nodes is private and only available to authorized people or services. In MetaCentrum we leverage the virtualization framework to setup virtual clusters overlaying physical resources. Using the framework it is possible to instantiate virtual workernodes connected to the dedicated private VLAN. The VLAN itself is accessible from all main NGI sites through the CESNET backbone network. Cluster members can thus be randomly allocated on all major NGI sites[13, 2]. Proxy servers are used to provide acces to the cloud services running inside such network (see fig. 4).

Establishing a cloud with multiple ElasticSearch nodes is as easy as starting an ElasticSearch instance on every node of the cloud. Utilizing the autodiscovery feature and automatic sharding, the nodes will establish themselves the topology and are ready to accept data to index and answer queries. No additional configuration of the nodes is necessary, especially no data schema needs to be specified. The data is just inserted and ElasticSearch handles it properly.

3.2 Turning logs into structured documents

Logs are not just text lines, but should rather be considered as structured data. There is at least a timestamp in every log along with some text data, and the data itself has a given structure[1, 12]. The structure of the log messages varies from product to product. ES is able to parse different

[4] `http://www.elasticssearch.org/`
[5] `http://www.kibana.org/`

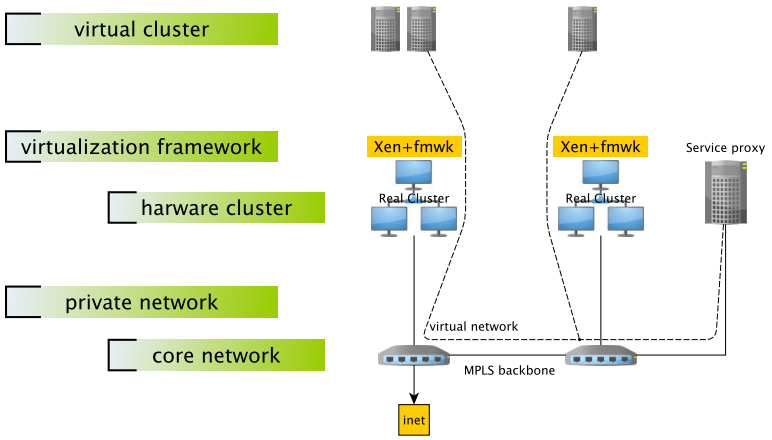

Figure 4: Private cluster

structuctures easily and index them so that queries can be made. Once the logs are indexed, it is possible to use standard ElasticSearch interfaces to ask quiries about them, which provides much better experience and response time compared with common tools like `grep`.

No special configuration is necessary on the side of ES, a default running ES instance is enough, regardless if it is a single node or a whole cloud. There are several ways how logs can be passed to ES for processing. The simplest way is to use the elasticsearch plugins available from the latest developement release of rsyslog. However, in order to keep a better control over the parsing process we decided to use Logstash[6] for several reasons:

Grok – the hearth of parsing. A library[7] which is able to parse text lines with regular expressions and turn them into structured data. Regular expressions are written in enhanced language which allows pattern reuse and provides ability to create nested rules in very convenient way.

Logstash flexible architecture – the concept of event/message processing pipeline consists from the set of input, filtering and output plugins working in the separate threads and interconnected with the sized queue. Logstash supports large scale of input and outputs which can be used to create flexible indexing services[8].

Logstash provides an entry point used insert the syslog data to ElasticSearch. The final arrangement is depicted in Fig. 5. *TODO: popsat strukturu (proxy + vc nodes, LS parser)*

From the experiments we have learned that logs sometimes contain binary data even when they should not[9]. There is no filter for those inside logstash itself so our solution does the filtering with a simple server consisting of unix tools `ncat` and `tr` (see MetaCentrum REPO).

3.3 Performance tuning

Establishment of the whole infrastructure is quite easy and can be done quickly. While the infrastructure works well for small datasets, it has performance limits which needs to be addressed before big data is processed.

[6] `http://www.logstash.net`
[7] `http://code.google.com/p/semicomplete/wiki/Grok`
[8] `http://www.logstash.net/docs/1.1.9/life-of-an-event`
[9] exception "source sequence is illegal/malformed utf-8"

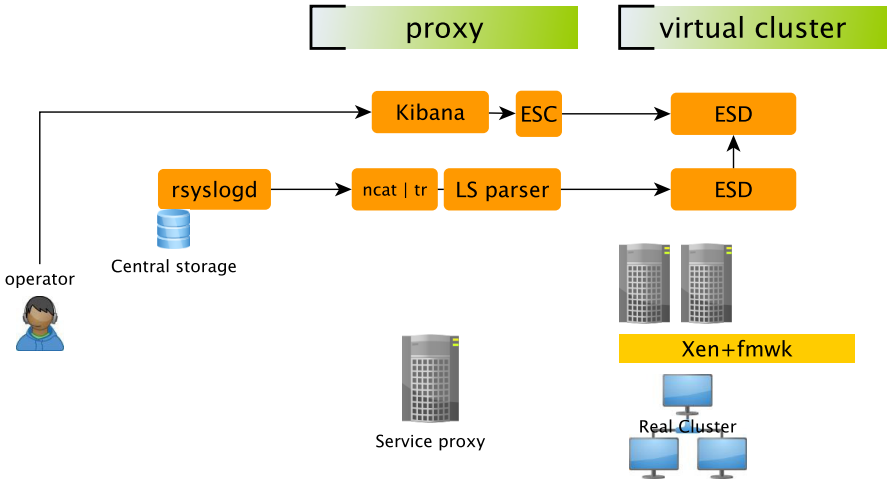

Figure 5: Proxy parser

3.3.1 Cloud parser

If more power is needed than what the existing ElasticSearch deployment provides, a larger cloud service can be instantiated. In a larger cloud, however, the single LS parser becomes a bottleneck which needs to be addressed, ideally by moving them closer to the indexing nodes. For that, some form of data spooling or other messaging queue is required, so indexing services running in the cloud can be configured to pop data from a single place. For example the `Redis`[10] server can be used in this manner[11], see Fig. 6.

Redis is populated with logs sent from the syslog server and provides the data to multiple LS parsers running in the cloud. Also it can serve as a buffer during ES cluster maintenance or high load. In advanced deployments there can be multiple redis queues running inside the cloud to provide failover and achieving more resilient environment.

3.3.2 Batching (Bulk indexing?)

The default behavior of most logstash input and output plugins is to process only one message/event at a time. That may be fine for low traffic environments but in cases of bootstrapping the ElasticSearch cluster, indexing historical dataset or recovering after total cluster failure, it is necessary to use some form of batch processing to enhance the uploading speed.

Redis input plugin Using `redis` as a message queue, batching can be done using pipelining of redis commands or using LUA scripts. Scripts can result in less network utilization while pipelining leads in the smoother client requests interleaving on the server side. The latest logstash implementation uses LUA scripting.

Elasticsearch output plugins – For the final delivery of structured data from logstash to ES cluster there are two main options:

- native api plugin – plugin does not use bulk indexing API yet, there is only a simple mechanism for having many inflight requests. This approach could create many indexing threads on the ES side which can lead to heap OOM.

[10] `http://redis.io/`
[11] `http://www.logstash.net/docs/1.1.9/tutorials/getting-started-centralized`

- http output plugin – In non batched mode every message is submited into ES by separate POST request and that can present significant overhead for standard syslog messages (avg. 150b). Our solution uses customized output plugin[12] implementing batching system with synchronized internal buffer queue and time based flush based on community proposals. There is also ongoing effort to implement general batching/flushing system into logstash.

3.3.3 Logstash Proxy vs. Rediser

TODO: zakl. myslenka: "redis frontu lze plnit pres LS proxy, ale to ma limity"

Logstash itself is meant to be an event processor which can serve on many places in logging infrastructure, e.g. as a shipper, central collector, indexing processor. Internal event processing and event objects themselves have some standard metadata associated with them (`source`, `source_host`, `source_path`, ...).

These various metadata fields would be populated usefully depending on plugins used (input file, input tcp, ...), but our deployment uses `rsyslog` as shipper (or `ncat` as replayer). All logs are collected elsewhere and forwarded by a single channel into indexing infrastructure so default metadata fields are in no real usage. In the case of ES bootstrapping they (and all code creating and working with them) present an unnecessary overhead. *TODO: odkazat prilohu s cisly?*

Logstash is written in ruby and so it is possible to elaborate on existing code and create lightweight skelet (rediser) which acts as simple proxy service and pushes incomming messages into redis queue without any modification. The parser can still use default LS redis input plugin which can work with plain messages poped from incomming queue, not just jsonified events.

Whole arrangement can be seen in Fig. 6.

Results from comparsion test can be found in appendix A.2.

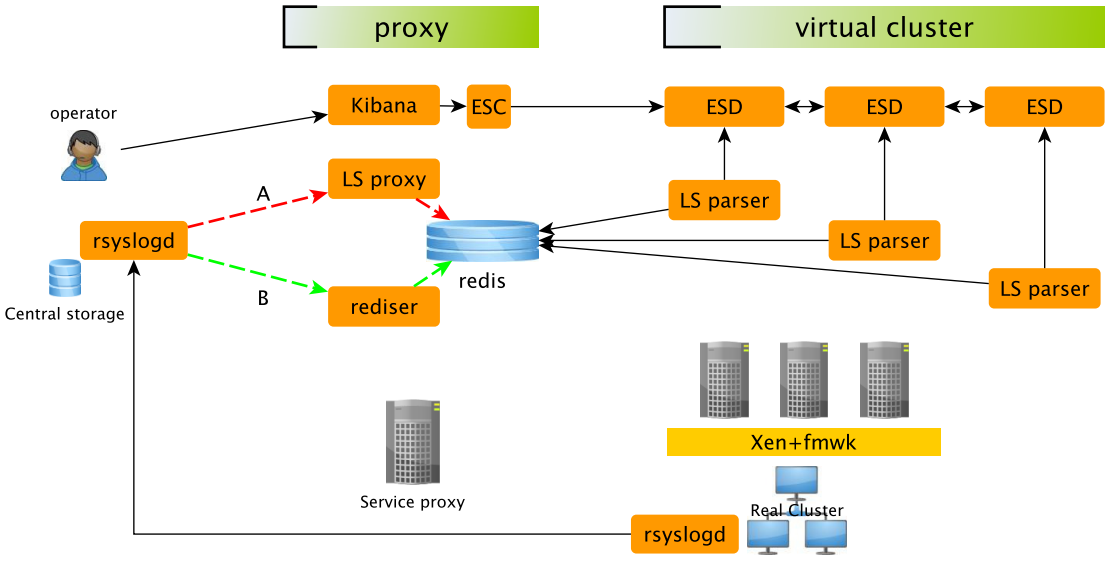

Figure 6: Cloud parser

3.4 The speed of ES upload and search

The speed of upload generaly depends on many factors:

- size of grokked data and final documents,

[12] see *MetaCentrum REPO*

- batch/flush size of input and output processing,
- filters used during processing,
- if multiple outputs are used in the single LS instance, the overall rate is limited by the slowest output plugin because of the internal queue between filters and outputs,
- elasticsearch index (template) setting.

A real dataset from January 2013 in the size of 105GB (797949196 events) was used for main testing. In the best found configuration the time for indexing is approximately 4 hours using 8 nodes ESD with 16 shared indexers(LS parser installed on all ESD nodes) and speed averages about 50 000 messages per second (eps). Tests results and detailed environment description can be found in appendix A.3.

During regular operations an incomming rate 100 - 5000 eps was observed. A single or two nodes cluster (type M) should be able to handle those rates, however this is yet to be prooved by long term production deployment. *TODO: lze delat na vyfouknutem clusteru*

Search speed was evaluated on basic level and is limited by IO throughtput, but even when using small cluster is much faster than simple `grep`, see Fig. 7.

```
# du -sh .
105G .
# time grep -R "realuser" * > search.txt
real 18m39.447s
user 1m7.796s
sys 1m24.565s
# wc search.txt
   81636  1773549 21777400 search.txt
#

# ./el_listnodes.py
10.0.0.31 id HRf5TJebQw6_cYxAsS2mtQ indices.docs.count 388345192
10.0.0.3 id BBcHTUk9SkWxhynzoPafog indices.docs.count 409604004
10.0.0.1 id 1pTq45TKTGavwnZGKXPcfQ indices.docs.count 0
# time sh curltest.sh > search1
real 0m34.944s
user 0m0.536s

# grep '"_id"' search1 |wc
  81636  244908 3265440
```

Figure 7: Performance of grep vs. ElasticSearch query

4 Advanced data analysis

Logs contain a lot interesting information that can reveal security incidents or attacks very quickly. The logs can also be correlated with other sources of information provided a combined knowledge, which can also point to a particular issue. While ElasticSearch makes it possible to quickly search the logs, it is not quite suitable to provide advanced analysis and data-mining in the data. Therefore we decided to utilize Mongo DB[13] to store parts of the logs and provide additional processing of them. We choose a nosql databaze since we aggregate information from multiple sources, which vary a lot in the way how it is structured.

The database is populated from the central syslog server in a way that is similar to feeding data to ElasticSearch. Unlike the feed to ElasticSearch however, only a fraction of data is stored in Mongo DB. For a pilot solution we focus mainly on information logged by the SSH server and we used the logstash and grok to parse authentication logs – the ruleset in Fig. 8 produces turn log line from Fig. 10 to events depicted in Fig. 9.

Statistical analysis and other analytic tasks can be done on the index/collection produced by such a `grok` filter. For example, the aggregation framework of MongoDB and its map-reduce

[13] `http://http://www.mongodb.org/`

```
AAARESULT (?:Accepted|Failed|Authorized|identification|Invalid|disconnect|tried|refused)
METHOD (?:[a-z~]+|correct key)
PRINCIPAL [a-zA-Z0-9_/~]+@%{HOSTNAME}

AUTHN %{AAARESULT:result} %{METHOD:method} for (invalid user )?%{USER:user} from %{IPORHOST:remote} port %{POSINT} ssh2
AUTHZ %{AAARESULT:result} to %{USER:user}, krb5 principal %{PRINCIPAL:principal} \(krb5_kuserok\)
SCAN Did not receive %{AAARESULT:result} string from %{IPORHOST:remote}
INVALID %{AAARESULT:result} user %{USER:user} from %{IPORHOST:remote}
DISCONNECT Received %{AAARESULT:result} from %{IPORHOST:remote}: 11: disconnected by user
WRONGKEY Authentication %{AAARESULT:result} for %{USER:user} with %{METHOD:method} but not from a permitted host \(host=%{IPORHOST:remote}, ip=%{IPORHOST}\)
REFUSED %{AAARESULT:result} connect from %{IPORHOST:remote} \(%{IPORHOST:remote}\)

SSHATTEMPT (?:%{AUTHN}|%{AUTHZ}|%{SCAN}|%{INVALID}|%{DISCONNECT}|%{WRONGKEY}|%{REFUSED})

SSHBASE3 (%{SYSLOGTIMESTAMP} (%{IP:coll} )?%{SYSLOGHOST:logsource} )?%{SYSLOGTIMESTAMP:timestamp} (%{SYSLOGHOST} )?sshd(?:\[%{POSINT}\])?:
SSHLINE %{SSHBASE3} %{SSHATTEMPT:message}
```

Figure 8: sshd auth log grok patterns

```
{
   "_id": ObjectId("51379dd1e4b0fad32a766fa7"),
   "@timestamp": ISODate("2013-02-03T13:12:44.0Z"),
   "@tags": [],
   "@fields": {
     "coll": {
       "0": "ddd.aaa.bbb.ccc"
    },
     "logsource": {
       "0": "wknode23.sub.domain.cz"
    },
     "result": {
       "0": "Invalid"
    },
     "user": {
       "0": "prueba"
    },
     "remote": {
       "0": "218.85.135.29"
    }
  },
   "@message": "Feb 3 14:12:42 ddd.aaa.bbb.ccc wknode23.sub.domain.cz Feb 3 14:12:44 sshd[5216]: Invalid user prueba from 218.85.135.29\n",
   "@type": "ssh"
}
```

Figure 9: Grokked auth event

```
Feb 3 14:12:42 ddd.aaa.bbb.ccc wknode23.sub.domain.cz Feb 3 14:12:44 sshd[5216]: Invalid user prueba from 218.85.135.29
```

Figure 10: Sample log line

techniques can be used to generate time-based hierarchical aggregations[14]. Using these techniques it is simple to group events originated in the same period of time. Other similar aggregated views can be produced too, see Fig. 11 for the whole schema.

The refined data can be further used for reporting or help with investigation of security events. All users and/or services interracting with the monitored domain can be profiled with the collected data. For instance it is easier to detect SSH attacks by correlating the number of unsucesfull login attempts.

It is also possible to include information from other sources. Our solution currently consumes data from *IDS Warden* operated by CESNET-CERTS within the CESNET2 network[15]. Warden itself is a messaging platform used for sharing data about security incidents and attacks that have been detected by the CESNET member institutions and also by externel partners. Data received from Warden is stored in MongoDB and used to detect attempts to access from malicious addresses, etc.

Because the volume of the data fetched through Warden can be large and installation of the client requires some non trivial software dependencies it was decided to host the client in the virtual cluster as well as MongoDB service itself.

Data collected in MongoDB can be processed directly using the MongoDB interfaces. However, the in order to simplify the operations and automate the most common queries we prepared a set of scripts to mine information from the database. These scripts can be run either manually on demand or as cron jobs. In the latter case the scripts emit alerts for security issues, like access from a attacking IP, etc. For instance, we provide tools to extract ....

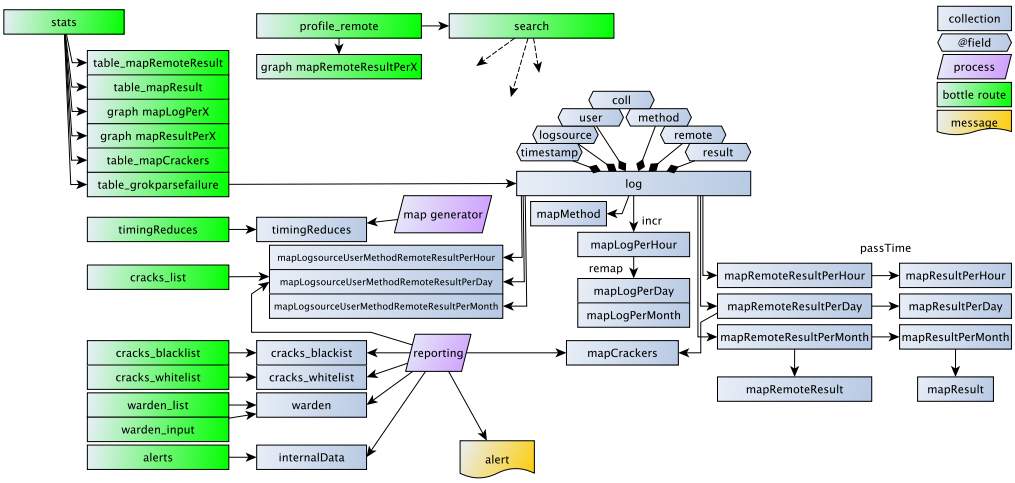

Figure 11: Mongomine schema

[14] http://docs.mongodb.org/manual/use-cases/hierarchical-aggregation/
[15] http://warden.cesnet.cz/

4.1 The speed of MongoDB upload

Logstash mongodb output does not use bulk inserts, neither we tried to develop this feature. Also only subset of sharding/clustering capabilites[16] of MongoDB was used during our experimentations. Mainly because MongoDB does not yet include any autodiscovery or autoconfiguration features.

A real dataset from January 2013 in the size of 20GB (33453234 events) was used for testing. Time for inserting all data is approximately 8 hours using 1 MongoDB server (type M) with 20 shards and 1 dedicated indexer with 6 LS instances. The average time of all incremental aggregations during normal operations is 10 seconds.

5 Overall schema of mining architecture

TODO: radsi taky priloha?

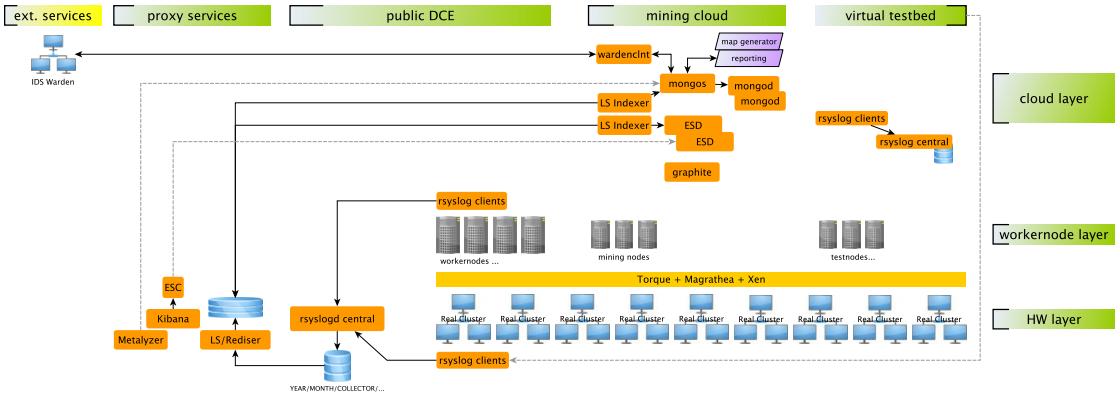

Figure 12: Overall schema of mining architecture

The final solution consists of:

- Build, test and deployment components
  - debian packages created from patched git sources forked from oss repository, tested in virtual testbed
  - rsyslog workernodes with disk assisted queues forwards messages through GSS-API secured channel
  - rsyslog central is stores all data on the disk and forwards them into indexing services
- Indexing and searching services
  - proxy spools incoming data for indexers in then queue
  - indexing services are running in the virtual cloud popping their workload from the queue
  - proxy provides access to searching services

[16] single host, 1 shard per CPU, 0 replica set

  - proxy provides support services to the cloud (dhcp, ...)
- Reporting and monitoring tools
  - environment is orchestrated from proxy server through set of shell and cron scripts
  - all components are installed from custom self-contained software packages provided to the cloud by the proxy
- Archiving
  - old logs are archived in a second tier storage – the Cesnet Storage Service[17]

Most of the used software and custom components can be found in *MetaCentrum REPO*, other data can be provided upon request if approved by CESNET-CERTS.

The system was developed mainly toward intrusion detection and incident response usecases. An example of an incident report generated by the system and its follow-up is described in Appendix A.4. In the presented example an alert generated from MongoDB reporting tool requires handler on duty to investigate such alert. System described in this paper and PoC of analytic application can ease handler with the process.

For this particular case the difference is that operator have learned all informations in 10 minutes using presented indexing and searching services. In contrast of using traditional log handling processors like grep or perl, which would take seriously much more time.

Described system is in production for about a year and in our current deployment we have 90% coverage of worker and service nodes connected to central logging service (approx. 750 nodes). System is also receiving logs from CERIT-SC cluster through single GSS-API enabled proxy.

The ammount of stored logs is approixmately 100GB per month. Using 7zip's most aggresive compression the same data can be compressed to 2GB.

TODO: starnuti/"rotovani" logu?

5.1 Lessons learned

The most important place in proposed architecture is the proxy. The performace of the whole system is limited by its capabilities. The services should be pushed into the cloud as much as possible (warden, graphite, ...). In advanced scenarios it would be necessary to provide more redundant proxy services.

Features provided by Metacentrum's virtualization framework (VMs and networking[**?**, **?**]) are very helpfull in service development.

Cluster management can be done using simple shell scripting but wider deployment will require refactoring or replacement with other configuration management tools[18].

We would like to develop more monitoring and trending services (IO and network monitoring) as well as more analytic, corelation and reporting functions.

Having experiences learned during this project it would be also possible to prototype other applications of tested technologies, for example Rob Lee's Supertimeline grok parser with embedded ES or FTAS or Warden database indexing and searching services.

Proposed system was tested as stated in this document, but longer production deployment will still be needed.

6 Acknowledgements

The access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005) is highly appreciated.

We also want to thank to the user groups of rsyslog (`rsyslog@lists.adiscon.com`) and logstash (`irc.freenode.org#logstash`) for their consultantcy support provided during our project.

[17] `https://du.cesnet.cz`

[18] https://github.com/electrical/puppet-logstash, https://github.com/electrical/puppet-elasticsearch

7 Conclusions

mereni se nam libilo

References

[1] Common Event Expression: Unified Event language for Interoperability.

[2] D. et al. Antoš. VirtCloud: Virtualizing Network for Grid Environments, 2009.

[3] Shay Banon. Road to a Distributed Search Engine , 2011.

[4] EMI. EMI Virt.

[5] Ruda M. et al. Scheduling Virtual Grids: the Magrathea System, 2007.

[6] R. Gerhards. The Syslog Protocol. IETF RFC 5424, 2009.

[7] Rainer Gerhards. Reliable Forwarding of syslog Messages with Rsyslog, 2008.

[8] LABAK. LABAK.

[9] F. Miao, Y. Ma, and J. Salowey. Transport Layer Security (TLS) Transport Mapping for Syslog. IETF RFC 5425, 2009.

[10] Peter Matulis. Centralised logging with rsyslog. Canonical Technical White Paper.

[11] Bodo Radoslav. `http://home.zcu.cz/~bodik/metasw/doc-harvesting-logs/pubdata`, 2013.

[12] Jordan Sissel. logging: logstash and other things - PuppetConf '12, 2012.

[13] Josef Verich Václav Novák, Pavel Šmrha. Deployment of CESNET2+ E2E Services in 2007, 2008.

A Appendix

A.1 Description of Published Data

http://home.zcu.cz/~bodik/metasw/doc-harvesting-logs/pubdata

All software provided in this package is provided "as is" and explicitly without any intention of releasing software tool of any quality. Most of the software presented in this report is present, other parts or data can be provided upon request if approved by CESNET-CERTS.

```
|-- cls                ... shell cluster management suite
|-- els                ... elasticsearch service
|-- gra                ... graphite service
|-- lsl                ... logstash proxy and indexer service
|-- metalyzer          ... mongomine PoC
`-- rsyslogd           ... rsyslog custom patches, build and test script
```

A.2 Comparsion LS vs Rediser

```
test description
---------------------
* time to push data into redis
* 8 ESD cluster M
* 16 shared indexers (MetaCentrum REPO)

proxy services
---------------------
A) LS
export JAVA_OPTS="-server -Xms1g -Xmx1g"
export JRUBY_OPTS="-Xcompile.invokedynamic=false"
-w1

input { tcp {
  host => "..."
  port => "..."
  type => "syslog"
} }
output { redis {
  data_type => "list"
  key => "qX"
  host => "127.0.0.1"
  ?batch => true
  ?batch_events => 1000
} }

B) ncat | tr | rediser:
 MetaCentrum REPO rediser

testdata description
  size: 286M
  messages/lines: 2000001
  rounds: 3

time to push all data into redis through logstash redis output
  test1 0:06:40
  test2 0:06:43
  test3 0:07:27
  total docs: 5999979 (+24 with binchars which does not pass :) = 6000003


time to push all data into redis through logstash redis output with batch 1000
  test1 0:01:54
  test2 0:01:52
  test3 0:01:50
  total docs: 5997700 (docs missing because of https://logstash.jira.com/browse/LOGSTASH-876)


time to push all data into redis through ruby1.9.1 rediser with batch 1000
  test1 0:00:25
  test2 0:00:24
  test3 0:00:24
  total docs: 6000003
```

A.3 ES upload tests

```
Environment description
-------------------------------
cluster M
model name      : Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
total cpus: 24
total mem: 22173MB
scratch size: 514G XFS
ESD -Xmx: 14782m


cluster N
model name      : Intel(R) Xeon(R) CPU           E5472  @ 3.00GHz
totral cpus: 8
total mem: 13989MB
scratch size: 43G XFS
ESD -Xmx: 9326m

proxy server
model name : Intel(R) Xeon(R) CPU            5160  @ 3.00GHz
total cpus: 4
total mem: 3993MB
redis maxmem: 1G

JVM Java(TM) SE Runtime Environment (build 1.7.0_10-b18)
Java HotSpot(TM) 64-Bit Server VM (build 23.6-b04, mixed mode)

Cluster X ESD 0.20.2
-Xmx (totalmem/3)*2
 -Xss256k -Djava.awt.headless=true -XX:+UseParNewGC -XX:+UseConcMarkSweepGC \
         -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly -XX:+HeapDumpOnOutOfMemoryError

Size of data
105GB in almost every case

Source of data
central
cloud node
Time to scp testdata: central > cloud node { 00:56:48

Data layout
ip based
time based

LS JAVA_OPTS
-server

LS wiring
dedicated indexer - indexers on separate machines
shared indexer - indexers deployed on ESD

LS batch params
redis input batch/es http output flush/filterworkes count

ES maint
wiped
deleted

fields template {"logstash":{"template":"logstash*","order":0,
"settings":
{"index.merge.policy_merge_factor":"XXX",
"index.number_of_replicas":"0",
"index.refresh_interval":"XXX",
"index.number_of_shards":"XXX",
"index.store.compress.stored":"XXX"},
"mappings":{
"_default_":{"properties":{"@fields":{"properties":{
"program":{"index":"not_analyzed","type":"string"},
"pid":{"type":"integer"}}},
"@timestamp":{"index":"not_analyzed","type":"date"},
"@tags":{"index":"not_analyzed","type":"string"},
"@type":{"index":"not_analyzed","type":"string"}},
"_all":{"enabled":false}}}}}

abort -- test was canceled
```

Figure 13: ES upload results

M	N	Source of data	Data layout	ES maint	Rediser flush	LS instances	LS params	LS _OPTS	LS wiring	No. shards	Refresh interval	Merge factor	Store compressed	Size of data	time to redis	shard size	total docs
0	16	central	ip based	wiped	1000	32	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	16	30s	30		105GB	07:00:56		
0	16	central	ip based	deleted	1000	32	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	16	30s	30		105GB	07:36:57		
0	16	central	ip based	???	1000	32	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	16	30s	30		98.4GB – abort	07:51:43		
0	8	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	16	30s	30		105GB	11:57:07		
0	8	central	ip based	deleted	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	16	30s	30		105GB	16:00:27		
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	30s	30		105GB	04:27:44	13GB	
8	16	central	ip based	deleted	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	30s	30		105GB	04:42:28	13GB	
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	30s	30		105GB	04:29:27	13GB	
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	30s	1000		105GB	04:31:08	13GB	
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	48	30s	30		98.4GB -abort	05:26:52	13GB	
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	-1	30		105GB	04:19:00	13GB	
8	16	central	ip based	wiped	1000	48	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	24	-1	30		105GB	04:17:07	13GB	797948858
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30		105GB	05:32:30	38G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30		105GB	05:28:55	38G	797948965
8	0	central	ip based	deleted	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30		105GB	06:14:05	38G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30		105GB	05:31:58	38G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:52:40	16G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:51:17	16G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:54:56	16G	797949196
8	0	cloud node	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:52:51	16G	797949196
8	0	cloud node	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:49:51	16G	798030881
8	0	cloud node	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:49:49	16G	797949196
8	0	central	ip based	wiped	1000	8	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	05:17:13	16G	797949196
8	0	central	ip based	wiped	1000	8	1000/4000/-w8	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	05:01:59	16G	797949196
8	0	central	ip based	wiped	1000	16	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	03:51:24	16G	797949196
4	0	central	ip based	wiped	1000	8	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	8	-1	30	TRUE	105GB	07:54:49	31G	797949196
4	0	central	ip based	wiped	1000	8	1000/4000/-w4	JRUBY_OPTS="-Xcompile.invokedynamic=false"	shared	4	-1	30	TRUE	105GB			

A.4 Real usecase

Alert (fig. 14) states successfull login from remote IP which has been blacklisted on one of the IDS datasource. Profiling this event on the user basis reveals that user is usually logging to the environment from the suspected network (fig. 15). All failed events are paired with successfull ones (password typos), but there is one suspicious event from a non usual remote ip. Using ES cluster allows operator to find probable cause of this event very quickly – at that time user account was expired and user was not retrying action (fig. 16). From examination of RIPE informations (fig. 17) on IP which triggered the alert and the user profile data, operator can conclude that either user's client machine is infected with some malware or someone else with infected computer is sharing the same addres from a NAT pool. That is also suggested by detailed Warden time information analysis (fig. 18).

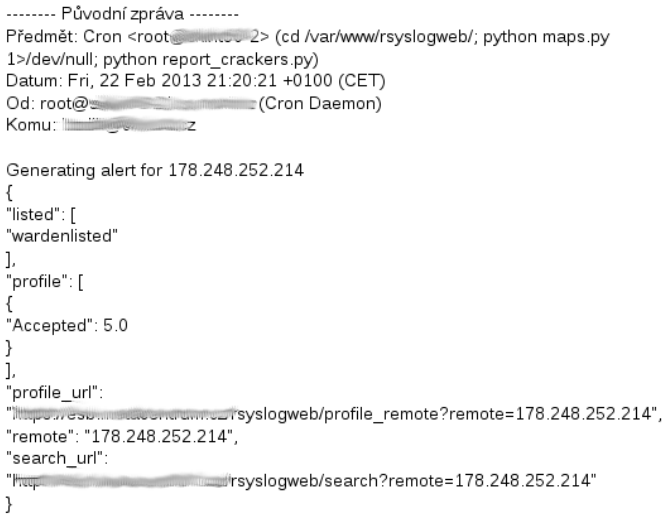

Figure 14: Mongomine alert

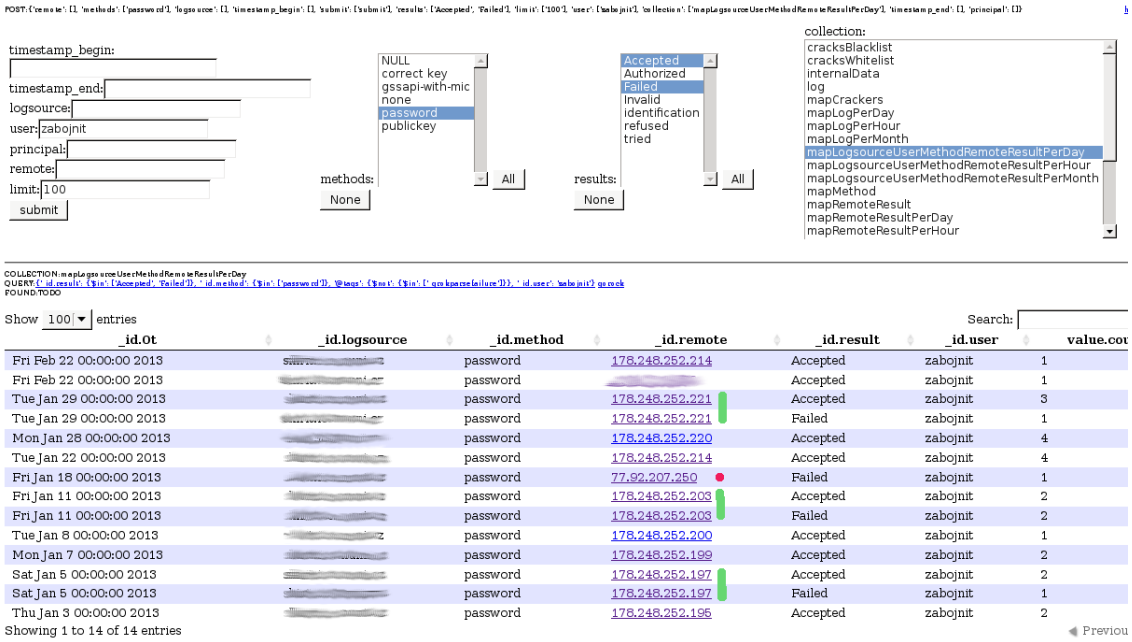

Figure 15: Profile userdata

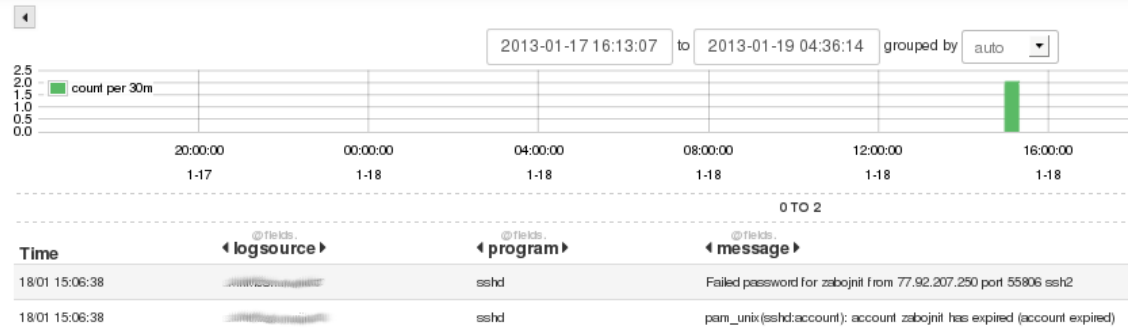

Figure 16: Kibana search

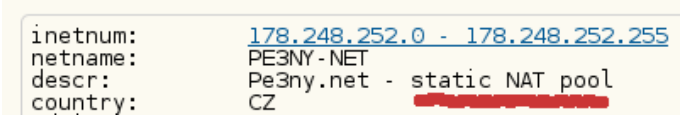

Figure 17: RIPE data

Figure 18: Warden event