PROCEEDINGS OF SCIENCE

Building a large scale Intrusion Detection System using Big Data technologies

Pablo Panero
CERN, Switzerland
E-mail: pablo.panero@cern.ch

Liviu Valsan*
CERN, Switzerland
E-mail: liviu.valsan@cern.ch

Vincent Brillault
CERN, Switzerland
E-mail: vincent.brillault@cern.ch

Ioan Cristian Schuszter
CERN, Switzerland
E-mail: cristian.schuszter@cern.ch

Computer security threats have always been a major concern and continue to increase in frequency and complexity. The nature and techniques of the attacks evolve rapidly over time, making their detection more difficult. Therefore the means and tools used to deal with them need to evolve at the same pace if not faster.

In this paper the implementation of an Intrusion Detection System (IDS) both at the Network (NIDS) and Host (HIDS) level, used at CERN, is presented. The system is currently processing in real time approximately one TB of data per day, with the final goal of coping with at least 5 TB / day. In order to accomplish this goal at first an infrastructure to collect data from sources such as system logs, web server logs and the NIDS logs has been developed making use of technologies such as Apache Flume [1] and Apache Kafka [2]. Once the data is collected it needs to be processed in search of malicious activity: the data is consumed by Apache Spark [3] jobs which compare in real time this data with known signatures of malicious activities. These are known as Indicators of Compromise (IoC). They are published by many security experts and centralized in a local Malware Information Sharing Platform (MISP) [4] instance.

Nonetheless, detecting an intrusion is not enough. There is a need to understand what happened and why. In order to gain knowledge on the context of the detected intrusion the data is also enriched in real time when it is passing through the pipeline. For example, DNS resolution and IP geolocation are applied to it. A system generic enough to process any kind of data in JSON format is enriching the data in order to get additional context of what is happening and finally looking for indicators of compromise to detect possible intrusions, making use of the latest technologies in the Big Data ecosystem.

International Symposium on Grids and Clouds (ISGC) 2018 in conjunction with Frontiers in Computational Drug Discovery
16-23 March 2018
Academia Sinica, Taipei, Taiwan

*Speaker.

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

PoS(ISGC 2018 & FCDD)014

1. Introduction

Since the 1970s when *Creeper*, which could be considered the first computer virus, and its killer counterpart, *Reaper*, appeared on the *ARPANet* [5], there has been a need to fight computer security threats.

There are two main categories of sources of logging information where traces of intrusions can be recorded and where detection can be performed. The first one is at the network level, meaning through a network connection to or from the target system. In this case a NIDS [6] can be used to monitor the network traffic looking for specific patterns or Indicators of Compromise to be able to react upon them, while still allowing the network to be used with minimum or no overhead at all. Both commercial and open source solutions such as *Snort* [7], *Suricata* [8] and *Bro IDS* [9] are readily available and can be used for this purpose. The second approach to detect intrusions is at the host level, where a HIDS can be used to monitor the activity and changes in the system files or running processes, for example using software such as *Tripwire* [10].

Regarding the internal workings of an IDS, as stated by Garcia-Teodoro et al. [11], there are three main techniques to detect intrusions: based on statistical behavior, using machine learning methods from categorized patterns and lastly by correlating with previously available knowledge. Design and implementations of all three types of IDS systems can be found in literature, as is the case of the Mahoney et al. [12] model to detect novel attacks by using statistical and machine learning methods or the previously mentioned *Snort*, *Suricata* and *Bro IDS* for the knowledge based type.

Nevertheless, the traditional knowledge based systems are starting to struggle to cope with the constantly increasing amount of network traffic and host data to analyze, and moreover not always making use of the full power that this amount of data can provide. Therefore, a new approach is needed. One option could be to use a DIDS, Distributed IDS, as proposed by Heberlein et al. [13]. Another point of view is taken by Marchal et al. [14], proposing a scalable NIDS that analyses DNS data, HTTP traffic and IP-flow records. This system is able to inspect and find anomalies based on signatures and the behavior information that can be seen in the traffic.

Taking similar steps as Marchal et al. a system that analyses several sources of data both at network and host level in order to detect intrusions has been implemented. Nonetheless, the system also takes advantage of data enrichment and gives more context on the intrusion so remediation actions can be taken in a more informed fashion.

2. The system

Figure 1 shows the overview of the system architecture, which is part of a bigger Security Operation Center (SOC). As it can be seen it is composed of four main stages: data ingestion and transport, data processing, storage / visualization and incident response.

2.1 Data Ingestion

The first step is that of data ingestion, in order to collect and send all relevant security data to the data processing component. Two classes or categories of data need to be collected and sent in order to provide both HIDS and NIDS capabilities. On one hand data is collected at

PoS(ISGC 2018 & FCDD)014

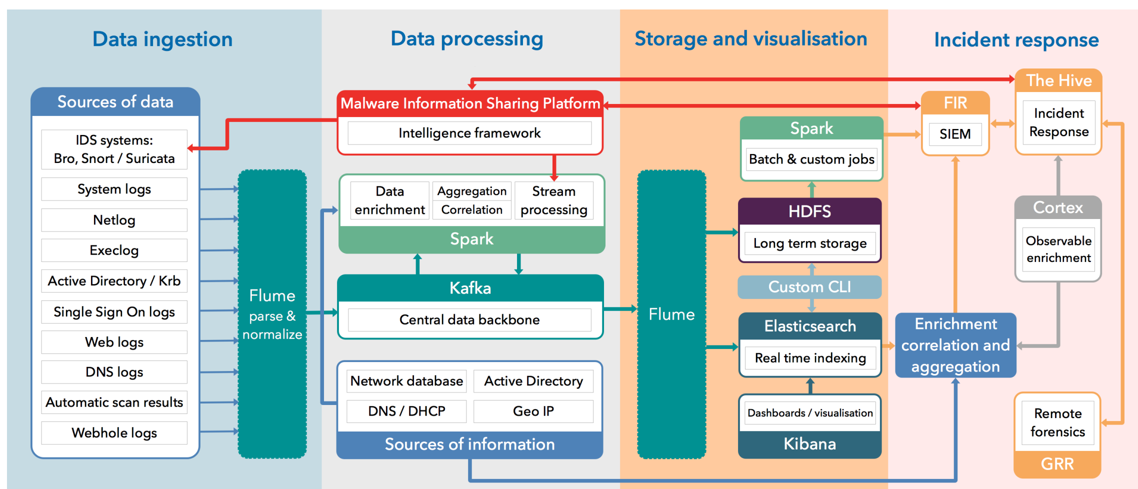

Figure 1: Top level architecture design

the host level using the *Execlog* module of the *Activity_klog* [15] project. This module logs all calls to the '`execve`' family of system calls, providing an effective way of tracking all executed commands. On the other hand, network data is collected from two sources. The first one, as in the previous case, is collected at the host level using the *Netlog* module of the *Activity_klog* project. The *Netlog* module logs all high level TCP and UDP activity by using the '`inet_stream_connect`', '`inet_dgram_connect`', '`sys_accept`', '`sys_close`' and '`sys_bind`' system calls. However, not all devices in the network run these *Activity_klog* modules, making any malicious activity related to them invisible in the previously collected data. Such is the case for the majority of the devices that are not centrally managed. Therefore, a traditional IDS, *Bro IDS*, is set up at the edges of the internal network logging the network traffic that crosses the perimeter. *Bro IDS* logs are also ingested. After the data from the previously mentioned sources is collected, it is sent to the central data buffer store, based on *Apache Kafka*. This task is carried out by *Apache Flume* agents located in each host that runs the *Activity_klog* modules and also in each of the *Bro IDS* nodes. These Flume agents push the data to a forwarding cluster, also based on *Apache Flume* agents, where parsing, normalisation and validation of data takes place. The forwarding cluster then sends the data to *Apache Kafka* topics.

Apache Kafka topics are further divided in partitions, which enable parallelism at the data transfer, storage and processing stages. Not every source of data has the same volume of incoming messages, and in consequence a higher or lower amount of partitions will be assigned to the different topics, based on the total amount of data / topic. In Table 1 a classification according to the volume of data, including average and peak amount of messages received per second by a topic and the amount of topics / partitions for each class of topics is shown.

PoS(ISGC 2018 & FCDD)014

Topic Type	Example	Avg. msg/sec	Peak msg/sec	No. of topics	Partitions / topic
Extra Large	bro_conn	8500	12000	1	40
Large	bro_files	5000	10000	2	20
Medium	bro_http	500	1500	10	10
Small	bro_ssh	100	300	7	3

Table 1: Topic classification by amount of messages

2.2 Data processing

The second stage of the system's pipeline is to process the data that was sent to the previously mentioned *Apache Kafka* topics. For this purpose, two separate types of *Apache Spark* jobs are executed, one for data enrichment and another one for intrusion detection. These jobs are developed using *Spark Structured Streaming*.

2.2.1 Data Enrichment

The first job takes care of enriching the data, in order to provide additional context. This job adds DNS information (hostnames where only IPs are present in the raw input data and IPs for the sources of data that only contain hostnames) as well as the GeoIP localisation (GeoIP) information. Due to the integration model of *Apache Spark Structured Streaming* and *Apache Kafka* the parallelism is at partition level. Executor-level concurrency had to be added in order to overcome the restrictions of one executor / partition, which led to logs being processed in a sequential manner by each executor and thus causing large amounts of lag.

The GeoIP queries' response time is negligible compared to DNS resolutions and the system can easily cope with such load. The case of the DNS resolution is far more complex. As shown in Table 2, DNS queries can be classified according to two properties: The class of the IP address and the result of the query.

Property	Types
IP address	Internal to the organisation
	External to the organisation
Query result	Resolved
	Not Resolved

Table 2: DNS Queries classification

The first category refers to the IP range that the corresponding address belongs to. It can be a local address, meaning it belongs to the organisation, or can be an IP not under local control. The second category refers to the result of the DNS query, meaning if a record exists (resolved) or not (not resolved). Every IP has an IP address category and a query result category resulting in four possible types, as shown in Table 3:

	Resolved	Not Resolved
Internal to the Org.	*A*	*B*
External to the Org.	*C*	*D*

Table 3: DNS Query types

PoS(ISGC 2018 & FCDD)014

In both cases of 'internal' addresses, *A* and *B*, the queries are answered fast enough for the system to digest the incoming data in real time. However, that is not the case for *C* and *D* addresses due to the recursive nature of the external DNS servers. This resolution might take between one and three seconds in the first case and between five and ten seconds in the second case. In some instances, like small or even medium sized *Apache Kafka* topics, the lag produced by *C* and *D* queries would be masked by the DNS resolution speed of the other types of addresses, resulting in an almost real-time output. However, in the case of medium sized *Apache Kafka* topics experiencing peaks conditions, large and even more in extra large topics, a delay of ten seconds in one query will generate a backlog of 10 000 or more additional addresses to resolve per second (for a total of approximately 100 000 in ten seconds).

The volume of queries to the DNS servers can become too high, in the orders of 6 000 per second, leading to load levels that the organisation's DNS servers are not able to easily cope with. By analysing the data, the reoccurrence of some addresses could be observed, and as a result a caching mechanism has the potential to greatly reduce the amount of queries down to the order of 900 per second [1].

Apache Spark jobs are executed in a cluster, each job being separated in two components: a driver, that controls the execution of the job, and executors, which carry out the data processing itself, in this case enriching the data. These executors are somehow volatile in the sense that they can be recreated in another node, or preempted if the scheduler (*Apache YARN* [16] this case) decides that the application does not need the amount of previously allocated resources. For this reason two different levels of caching are used. A first level of software caching is used and kept in the memory of each executor. This is a Least-Recently-Used (LRU) cache, meaning that when it is full it will evict the key that has been least recently accessed. In addition this keys have an expiration time or time-to-live (TTL) which means that value is not useful anymore once the expiration time is reached and in consequence these keys are evicted from the cache. However, this cache lives only while the executor lives. Once an executor is terminated on a given worker node and reinstantiated on a different worker node, the in-memory cache is lost. Therefore a second level cache is used on a separate, centralised, *Redis* [17] in-memory database. This centralised cache allows to quickly check for DNS resolutions that other executors might have already performed and also to fill up the local cache when an executor is created or reallocated. As it is the case for the local software cache, *Redis* is also configured to use a 'volatile-lru' expiration method along with a TTL for each individual key.

However, 'external' DNS resolutions pose a problem that potentially making the pipelines lag behind. One option to overcome the *C* and *D* resolutions would have been to increase the level of parallelism, by increasing the amount of partitions, but this can become prohibitive in terms of hardware needs and costs for the cluster used for executing the *Apache Spark* jobs. Two solutions are employed by our system in order to mitigate this issue.

The first solution is developing an asynchronous processing pipeline, for old long-running connections and abnormal bursts in network activity. A third *Apache Spark* job, which performs exactly the same processing as the enrichment job is set up. Nonetheless, this job will only execute queries for external addresses for which there are no DNS resolution results cached. This job re-

[1] Both measurements have been taken with only four topics: one small, two medium and one large

PoS(ISGC 2018 & FCDD)014

ceives about 200 messages per second on average and 1 000 under peak load situations. Therefore, even if these resolutions generate some lag, it will only cause the asynchronous pipeline to lag behind, allowing the normal flow to keep the pace. Messages are sent to this processing workflow when the main pipeline is lagging for more than 30 minutes (configurable) and the address is of external type and not found in the cache. The results of these DNS resolutions are added to the centralised *Redis* cache so that the normal DNS enrichment processes have access to them.

The second solution consists of building the architecture of the jobs in a way that takes advantage of the concurrency at the level of each *Apache Spark* executor. Since the parallelism of *Apache Spark* and *Apache Kafka* are at the partition level, a pseudo-batching mechanism is employed by each of the executors, using the '`mapPartitions`' function on the streaming *DataFrame*. A configurable number of rows are fetched into memory by each executor, using a grouped iterator, which is lazily iterated by default and ensures that the job is non-blocking. All fields that need to be resolved are collected in unique sets of homogeneous elements (e.g. IPs or hostnames). Then, a second configuration parameter dictates how many IPs or hostnames from the batch should be resolved concurrently, creating a number of *Scala Futures* that are collected once they complete. Using the results from the *Futures*, a second pass is done through the input data, enriching the logs. A work-stealing pool [26] from Java 8 is used in order to ensure that the main thread pool running *Apache Spark* is not affected and that the multi-threaded computations are executed as efficiently as possible. Through this approach, we are able to resolve the hostnames and IPs in parallel, mitigating most of the issues. The approach works because the workload of each *Future* is minimal, since most of the time is spent by waiting either for data in *Redis* or responses from the DNS servers. The sizes of the batches being resolved concurrently can be adjusted as much as the hardware running the other services (*Redis* and DNS servers) allows it. In Figure 2 the full process of enrichment is shown.

2.2.2 Intrusion Detection

The second job takes care of performing the intrusion detection process, correlating the given indicators of compromise with the data, and creating a notification message in case of a positive match. The implementation of this IDS provides additional flexibility with regards to the use of threat intelligence for the purpose of detecting intrusions. Moreover, due to the underlying technologies from the *Apache Big Data* ecosystem, this IDS is able to scale out on demand, a feature that *Bro IDS* lacks. If needed, due to scaling problems, this job could potentially substitute the intrusion detection component of *Bro IDS*.

This job also analyses data from 'Execlog' and 'Netlog' data sources in order to provide HIDS and NIDS capabilities respectively. The data is consumed from enriched *Apache Kafka* topics. The appropriate fields matching the types of available IoCs (e.g. IPs, file hashes, URLs, etc.) are being analysed. These Indicators of Compromise are stored in a *MISP* instance and fed periodically to this intrusion detection *Apache Spark* job. As in the previous case, the amount of data or IoCs to correlate and look for can exceed the capabilities of the cluster. Meaning that it would not be feasible to keep it in memory, and in addition the cost of lookups in a data structure such as a hash map might not be affordable for a large amount of keys. Therefore, *Bloom Filters* [18] are used, meaning that only up to K positions of the filter will be queried in each lookup, where K is being configured depending on the nature of the IoC (IP, Hash, URL, etc.) and the number of

PoS(ISGC 2018 & FCDD)014

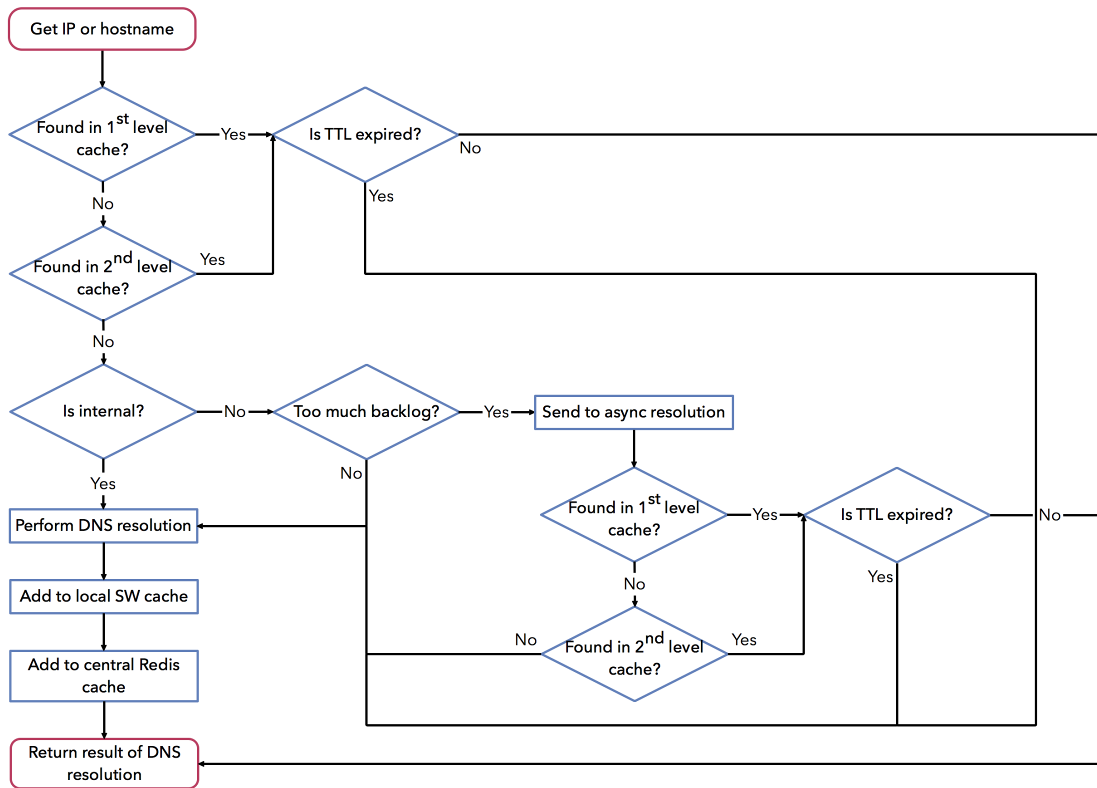

Figure 2: DNS data enrichment flowchart

IoCs of each type. Nevertheless, since a *Bloom Filter* can produce false positives (but not false negatives) a second stage of checks needs to be performed to avoid false positives in intrusion detection notifications. For this matter, *Redis* is used again, serving as the source of truth, giving no false positives.

Finally, if there is a hit, the message along with some context (i.e. elements that produced the hit and what IoC was detected) will be pushed to a specific *Apache Kafka* topic.

2.3 Data Storage and Visualization

Once the data has passed through the processing pipeline it follows two paths. The output of the data enrichment job gets stored in *HDFS* [19], for long term storage, and *Elasticsearch* [20], for real time indexing, again using *Apache Flume*. The detection job only emits an output if there is a positive match, with the data being stored in *Elasticsearch*.

Lastly the data, including the intrusion notifications, can be visualized using *Kibana* [21], which makes use of the data stored in the underlying *Elasticsearch* instance.

To sum up, the full *Apache Spark* job architecture is shown in Figure 3.

PoS(ISGC 2018 & FCDD)014

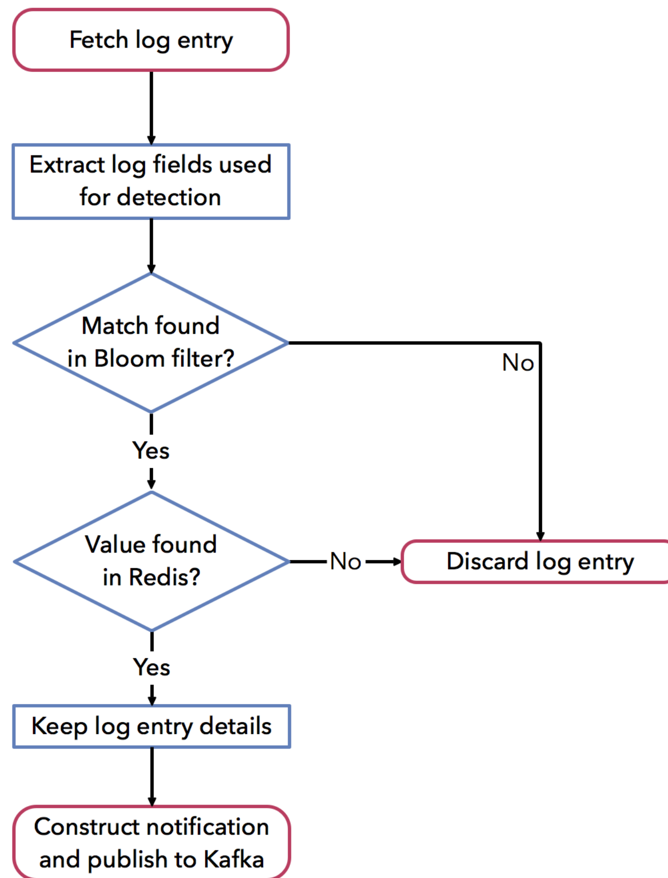

Figure 3: Intrusion Detection flowchart

3. Validation and Monitoring

So far, the design for a functioning system has been described. However, there is a need to know in a precise manner that the jobs are running properly. In order to fulfil this requirement a monitoring system has been implemented. This monitoring system consists of two *Collectd* [22] plugins that get measurements of the *Apache Kafka* topics and *Apache Flume* consumer groups, *Grafana* [23] dashboards for visualization, and finally a series of *Apache Spark* jobs that are run by a scheduler, *Nomad* [24]. The *Nomad* monitoring jobs are the following:

- *Apache Spark* jobs are run in client mode, which means that the driver runs on the machine that submits the job to the cluster. Since these jobs are launched by the *Nomad* scheduler, the drivers run on the *Nomad* worker nodes. This allows *Nomad* to monitor and relaunch the jobs if they fail or are stopped for unexpected reasons.
- Monitoring of the lag of each enrichment *Apache Kafka* topic, as well as its source, because if the data in the source *Apache Kafka* topic is delayed so will be the data in the enriched *Apache Kafka* topic. Moreover, the comparison between the latest message both in the source

PoS(ISGC 2018 & FCDD)014

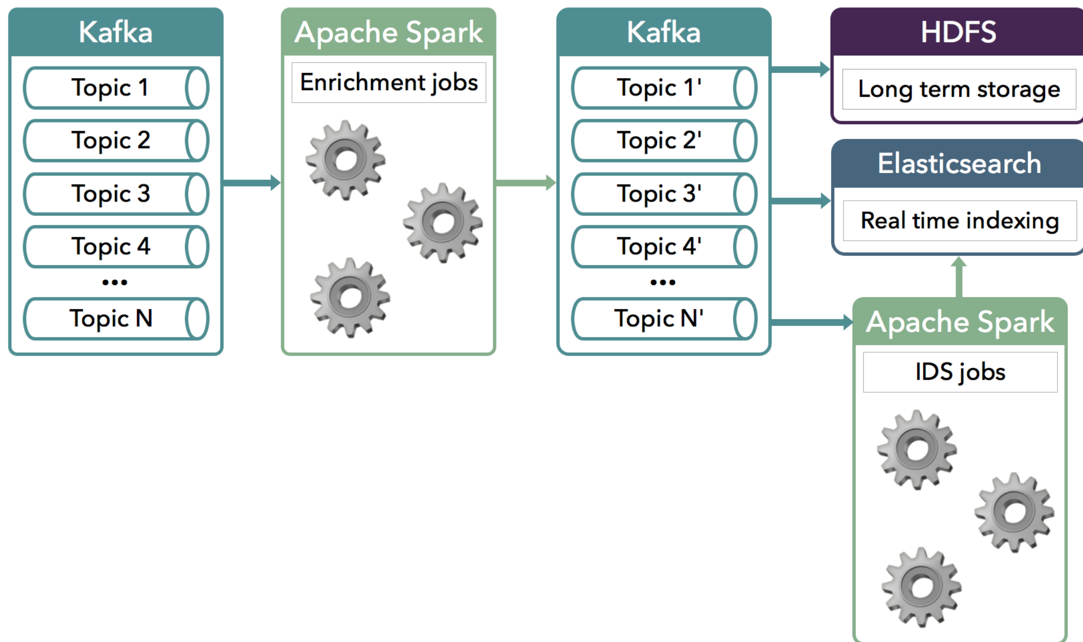

Figure 4: Technological architecture

and in the enriched data is monitored. Therefore, we can detect the lag independently in the source and the enriched *Apache Kafka* topics, but also in between them. In Figure 5 an example of the timestamp and volume difference between source and enrichment *Apache Kafka* topics is shown.

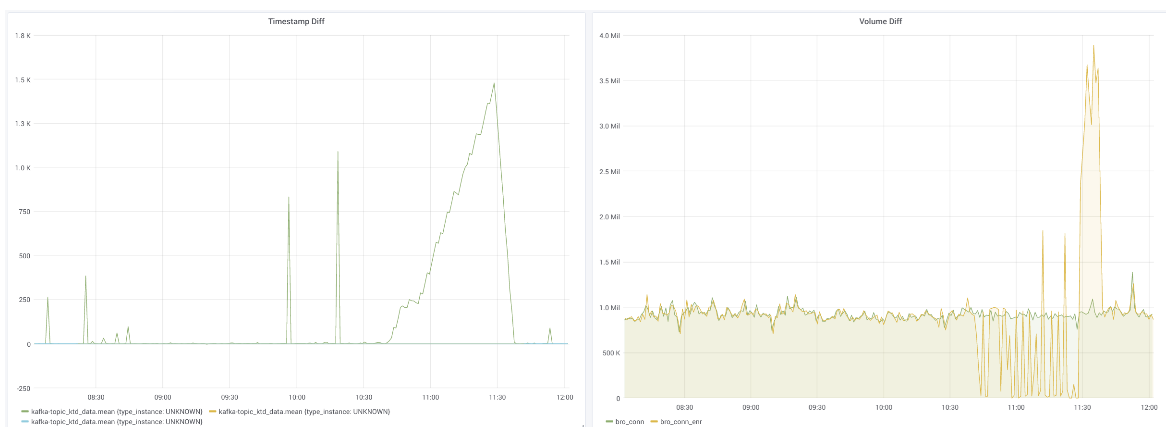

Figure 5: Timestamp difference and volume monitoring

- As mentioned above, there is a third job, which performs asynchronous DNS resolutions for entries that are not already in the cache so the main processing pipeline does not continue to increase the backlog. Therefore, the lag between the current timestamp and the last message

PoS(ISGC 2018 & FCDD)014

timestamp in each of the data sources (*Apache Kafka* topics) is compared. In addition, a dummy message to be enriched is sent directly to the asynchronous topic without passing through the normal source *Apache Kafka* topics. The enriched message is then sent to a specific topic from which a monitoring job is reading and computing the lag.

- The data storage has to be monitored in order not to lose data. In consequence, the backlog and the throughput of both sets of *Apache Flume* agents, writing to *HDFS* and *Elasticsearch* respectively, are monitored. In Figure 6 an example of the lag and throughput monitoring is shown.

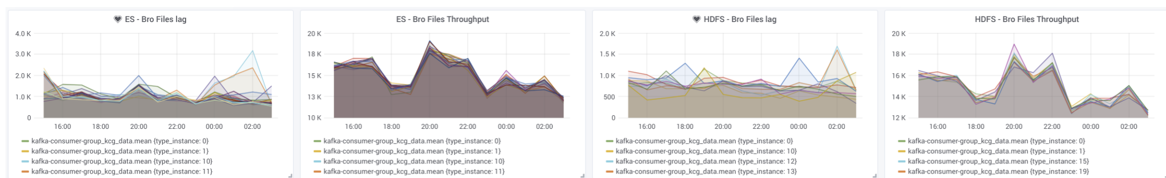

Figure 6: HDFS and ES lag and throughput monitoring

- The *Redis* instance is monitored, measuring the amount of open connections, memory usage, evicted and expired keys.

4. Conclusion

The system was proven to work and produce the same type of alerts as *Bro IDS*, and even more given that it can be fed with more data from other sources.

Nonetheless, the system is currently running only on 50% (500 GB / day) of the expected data in the enrichment phase and on 12% (500 GB / day) of the data in the detection phase. Since it is running on technologies such as *Apache Spark*, *Nomad* and *Apache Kafka* it should be trivial to scale the system out.

5. Future Work

As mentioned before the *Apache Spark* gives more flexibility than *Bro IDS*. Therefore, another special job could be added to detect well-known ad-hoc patterns of attack (e.g. binaries being executed from locations where malware frequently gets written to following an intrusion) that cannot be detected by a single Indicator of Compromise in *MISP*. The same can be used via data grouping (i.e. network and host) and making use of the latest MISP objects [25] which allow the user to correlate multiple related Indicators of Compromise.

In addition, Machine Learning and Anomaly Detection capabilities are currently being developed.

Moreover, the *Redis* database is not running in high availability mode, which makes it a single point of failure. Therefore, deploying *Redis* in cluster mode would be desirable.

Another point to take into account concerns the GeoIP databases. Since they are static they need to be periodically downloaded from the GeoIP data provider. These databases need to be

PoS(ISGC 2018 & FCDD)014

redeployed in the monitoring nodes (*Nomad*) and followed by a restart of the *Apache Spark* enrichment jobs so that the updated databases get propagated to the executor nodes. One possibility would be to create a centralised service that processes the queries and acts as a proxy to these GeoIP databases.

References

[1] Apache Flume. https://flume.apache.org/

[2] Apache Kafka. https://kafka.apache.org/

[3] Apache Spark. https://spark.apache.org/

[4] Malware Information Sharing Platform. http://www.misp-project.org/

[5] CHEN, Thomas M.; ROBERT, Jean-Marc. The evolution of viruses and worms. Statistical methods in computer security, 2004, vol. 1.

[6] MUKHERJEE, Biswanath; HEBERLEIN, L. Todd; LEVITT, Karl N. Network intrusion detection. IEEE network, 1994, vol. 8, no 3, p. 26-41

[7] Snort IDS. https://www.snort.org/

[8] Suricata IDS. https://suricata-ids.org/

[9] Bro IDS. https://www.bro.org/

[10] Tripwire. https://www.tripwire.com/

[11] GARCIA-TEODORO, Pedro, et al. Anomaly-based network intrusion detection: Techniques, systems and challenges. computers & security, 2009, vol. 28, no 1, p. 18-28.

[12] MAHONEY, Matthew V.; CHAN, Philip K. Learning nonstationary models of normal network traffic for detecting novel attacks. En Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2002. p. 376-385.

[13] HEBERLEIN, L. T.; MUKHERJEE, B.; LEVITT, K. N. Internetwork security monitor: An intrusion-detection system for large-scale networks. En Proceedings of 15th National Computer Security Conference. 1992. p. 262-271.

[14] MARCHAL, Samuel, et al. A big data architecture for large scale security monitoring. En Big data (BigData Congress), 2014 IEEE international congress on. IEEE, 2014. p. 56-63.

[15] Activity Klog project. https://github.com/CERN-CERT/activity_klog

[16] Apache YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[17] Redis. https://redis.io/

[18] GUO, Deke, et al. Theory and network applications of dynamic bloom filters. En INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. IEEE, 2006. p. 1-12.

[19] Apache HDFS. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[20] Elasticsearch. https://www.elastic.co/

[21] Kibana. https://www.elastic.co/products/kibana

PoS(ISGC 2018 & FCDD)014

[22] Collectd. https://collectd.org/

[23] Grafana. https://grafana.com/

[24] Nomad. https://www.nomadproject.io/

[25] MISP Objects. https://github.com/MISP/misp-objects

[26] Blumofe, Robert D and Leiserson, Charles E, Scheduling multithreaded computations by work stealing, Journal of the ACM (JACM). volume 46. pp. 720-748.

PoS(ISGC 2018 & FCDD)014