Cloud-Based Protocol IDS using Apache Kafka and Spark Streaming

Leon Wirz
Sirindhorn International Institute of Technology, Thammasat Univerity, Pathum Thani, Thailand
leon.wir@dome.tu.ac.th

Rinrada Tanthanathewin
Sirindhorn International Institute of Technology, Thammasat Univerity, Pathum Thani, Thailand
rinrada.tanth.gal@gmail.com

Asipan Ketphet
Sirindhorn International Institute of Technology, Thammasat Univerity, Pathum Thani, Thailand
asipanketphet@gmail.com

Abstract*—In the current era of technological advancements web applications are a major medium in the internet world. Many of these mentioned web applications adopt the usage of a dedicated back-end server where most of the technical processes happen. Typically the front-end sends requests to the back-end which can be done in multiple ways including, GraphQL, gRPC, REST, etc. In this particular paper, the focus is on REST as it is still considered a norm in most web applications. In REST, HTTP requests by the front-end which are mapped to GET, POST, PUT, and DELETE have been proven to be prone to attacks and exploits that can inflict extensive amounts of damage to a system. Therefore, this paper's main point will be the detection of such attacks and exploits, mainly being Automated Brute Forcing on web-based login, HTTP flood attacks, SQL injections (SQLi), and Cross-Site Scripting (XSS). Both Apache Kafka and Spark streaming are used as main pillars in the architecture for processing the user inputs in REST HTTP header fields, which are the main vulnerabilities that are being exploited in such attacks. As HTTP requests from the front end of a web application come in massive volumes, it is crucial to process these HTTP requests in an efficient and resourceful way. Spark streaming allows both of these critical processing characteristics by using batch processing and MapReduce. Apache Kafka allows back-end engineers to send real-time request logs to a Kafka broker and furthermore bridges the streamed data to Spark streaming using the Spark Streaming & Kafka Integration library. In this paper, algorithms that were developed in Scala are used to determine if an HTTP request contains any patterns that resemble any of the 4 mentioned attacks and both send an alert to the system security engineer and log the event in a Google Cloud Storage Bucket, as the developed IDS is implemented on Google Cloud Platform.

Keywords—Intrusion Detection, Apache Kafka, Spark Streaming, SQL injection, Cross-site scripting, Brute Force Login, HTTP flooding

I. Introduction

An Intrusion detection system is a security system which main functionality is the monitoring and analysis of system events for the purpose of finding and providing real-time or near real-time alerts and warnings of attempts of unauthorized access to system resources. Protocol-based intrusion detection systems are specialized IDSs that are commonly installed on web servers and used to exclusively monitor and analyze HTTP or HTTPS requests in a stream-like format. There are three types of intrusion detections methods, which are as follows:

i) Signature/Heuristic-based detection, which is a simple and efficient method of detection that involves the matching against large collections of rules or known patterns of malicious data. Nevertheless, the ability of an IDS that uses this method to correctly detect an intrusion solely depends on the available collection of signatures. Hence, the system cannot detect unknown intrusions in which patterns are not contained in the collection of signatures.

ii) Anomaly-based detection, which involves actual observations is more dynamic and more intelligent compared to the latter. Using either statistical methods or Machine-learning are used to analyze events and categorize them into either being a legitimate request or an intrusion attempt. Statistical methodologies are applied to observe the behavior of events using univariate, multivariate, or time-series models to observe measurable metrics. The Machine-learning approach uses Classification ML models, which work by intaking events and activities and classifying them into being either intrusion attempts or not, as has been done in [3]. Additionally, more advanced ML models can classify events into more than two classes as in [11], where Siva Reddy and Saravanan have developed multiple Classification Models with more than 2 variations of outputs using one-hot encoding. Regardless of the mentioned advantages, compared to Signature/Heuristic-based detection, Anomaly-based detection's performance is not as favorable and has some negative effects on less powerful machines, running real-time IDS.

iii) Distributed/Hybrid-based detection, which is performed using a combination of Signature/Heuristic-based detection and Anomaly-based detection. This detection method has a more complex structure, but with it comes a better identification rate and response time.

A good intrusion detection system including protocol intrusion detection systems should give results in a real-time manner, must not take up a lot of resources, have a low false positive and false negative rate, have high to permanent uptime, and cover a wide area of intrusion methods that it can detect. But for protocol intrusion detection systems, it is difficult to achieve the characteristic of not using up a lot of resources because medium-sized to mainstream-sized web applications have a massive amount of HTTP requests that are sent at all times. Applying any kind of algorithm efficiently has always been a challenge, which this paper will be tackling. Furthermore, the problem of having permanent uptime on an on-premise IDS is not guaranteed, as it is an apparent problem that comes with every system that is on-premise due to power or internet outage, natural disasters, etc.

This paper's IDS is a cloud-based protocol IDS, which means that it is a protocol IDS that runs on a cloud environment, specifically Google Cloud Platform (GCP) [21]. This solves the problem of the lack of guarantee when talking about the uptime which should be high enough to a certain degree. Google Cloud Platform's service level agreement (SLA) dictates that the services that the system uses have a corresponding uptime being:

i. Compute Engine single instance SLA Monthly Uptime percentage of $\geq 99.5\%$

ii. Cloud Storage in regional location SLA Monthly Uptime percentage of $\geq 99.9\%$

iii. Dataproc SLA Monthly Uptime percentage of $\geq 99.5\%$

This allows the system to run almost permanently as the lowest SLA is 99.5% per month. The problem of huge amounts of streaming data, which needs to be processed in an efficient way, which comes with being a Protocol IDS is also being approached with the combination of using Apache Kafka [20] and Spark Streaming [19]. Kafka acts as the receiving end of a bridge that allows all the streaming data to pass through via four separate topics in a broker, being GET, POST, PUT, and DELETE, which correspond to their actual HTTP method name. These four topics, which are streaming data are then sent to separate Spark Streaming jobs that each detect their own Intrusions namely:

i. *Automated Brute Forcing on web-based login*

ii. *HTTP flood attacks*

iii. *SQL Injections (SQLi)*

iv. *Cross-Site Scripting (XSS)*

Spark Streaming is a framework that works on Spark; therefore, it utilizes parallel computation and synchronization of multiple machines on one or more clusters. Dividing Streaming data into batches by time frames allows the usage of batch processing, furthermore the creation of a specialized Data structure called Resilient Distributed Dataset (RDD). Algorithms created in Spark typically run MapReduce Jobs which have been proven to give tremendous amounts of resource utilization and efficiency, which can be backed by Tun's, Nyaung's, and Phyu's experimentation results in [12], which show that 500,000 records as one batch can be processed in around 0.5 to 1.2 seconds using medium computation power, being an Intel Core i5 processor and 8GB memory.

Automated Brute Forcing [1] on web-based login is the act of using a program or script to forcefully send login requests, typically GET requests in HTTP to a web server. The frequency of requests sent varies from program to program and machine sent from. The Automated Brute Force Login detection algorithm alerts and logs requests if a certain request limit threshold is breached from a single IP address attempting to send requests at a high frequency. HTTP flood attacks are similarly detected using a different algorithm. Since HTTP flood attacks originate from all four HTTP request methods, being GET, POST, PUT, and DELETE, all requests are being processed in the HTTP flood attack detection algorithm. The frequency counting part also remains the same with the exception of the request frequency limit which is dependent on the implementation of IDS itself. As for SQL injections and Cross-Site Scripting detection, both have algorithms that function closely. A list of sub-strings for each attack is constructed which are extracted from datasets [16] [17], which are most likely to occur in an attack attempt.

The main objective of this paper is to show the capabilities of Apache Kafka and Spark streaming when used as a Protocol IDS to detect the aforementioned attacks while describing the architecture and methods used in detail with their corresponding results. Moreover, this paper's architecture can be used as a foundation for more specialized implementation in the Future.

II. Related Work

In 2020 Hai and Khiem [4] proposed an architecture for processing IDS logs using Spark Streaming. The focus is to improve the performance of Network Intrusion Detection Systems (N-IDS) by using distributed processing and parallel computing with Apache Spark. Three distributed computing models are compared with each other, being:

i) *Distributed Snort*

ii) *HBase & Impala*

iii) *HBase & Impala & Additional VM.*

The computational efficiency is best in point iii), as HBase & Impala have been shown to have higher throughput, adding more VMs, which do not only increase computational power but also add additional space, therefore being more powerful. This inspired the usage of using a distributed system to efficiently process the great number of requests that are poured into the system.

This concept is then further built on by the presented in [12], which is used to evaluate the performance and has presented promising results when it comes to huge amounts of records. In this system 100 000, 300 000, and 500 000 records have been used as a testing load for a 1-interval time frame. These records are sent to Spark streaming via Apache Kafka, which this paper's architecture has fully adopted. The results of the experimentations indicate that performance issues do not occur even if the number of records fed into the system is immense. This allows the implementation of this paper's protocol IDS to be more realistic as many back-end frameworks have implemented Apache Kafka in their language and can be used to send streaming data from their web application to the IDS which is hosted on the cloud.

Furthermore, in the year 2021 Elmasry, Akbulut, and Zaim [14] proposed a design of integrated cloud-based intrusion detection systems (CIDS) using third-party cloud service. This design integrates all of the modules that are used in an IDS to be migrated and fully used in a third-party cloud environment, where the operations of Monitoring, Processing, Analysis, Prediction, and Response are accomplished on the cloud. This configuration's purpose is explained to have a more resourceful computation and higher uptime during the Processing and Analysis compared to a normal system that is on-premise. Elmasry's, Akbulut's, and Zaim's modules have been partially implemented and mapped into this paper's proposed system. The Monitoring Module is depicted as the Kafka broker, which receives data streams from the four producers depending on the HTTP request method. The Processing and Analysis Modules are mapped to the four Spark Jobs which contain algorithms to detect intrusions. The final module this paper implements is the Alert module which consists of an embedded Vonage API call for each Spark Job accompanied by the logs that are saved on a dedicated Google Cloud Storage bucket.

This paper's source code is heavily influenced by an Open-Source GitHub project made by Xu [15]. Visor which is a real-time log monitoring program using Kafka is used to log HTTP requests in a special format, which can be seen in Figure 1.

```
127.0.0.1 user-identifier frank
[10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif
HTTP/1.0" 200 2326
```

Figure 1 : Log Pattern of HTTP requests

This real-time log monitoring system is the basis of this paper's streaming data input and format. Apache Kafka takes in or digests these requests from producers to topics according to HTTP request methods and then relays them to consumers that specifically request these topics.

Debnath et al [10] have conceived a model for Real-time Log Analysis System called LogLens. The Dynamic Programming Algorithm is used to detect abnormal log sequences of an event or transaction. This Algorithm goes through different parts of a log's entry and checks important parameters against a deterministic value, String, pattern, or Wildcard. This idea is applied in this paper's algorithm for detecting SQL Injections and Cross-Site Scripting, where collections of Wildcards are used to compare patterns or signatures that fall under each category.

The detection mechanism is placed in a manner where the actual request is not forwarded to the system, as the requests can be first sent to the Kafka broker before actually sending any real requests to the back-end system, except for Cross-Site Scripting which is affecting the front-end part of the architecture of an application. This design choice has been proposed by Joshi, Ravishankar, Raju, and Rave [6] in 2017. Having the requests being processed and analyzed before relaying them to their actual destination increases the chance of taking action by another system or entity. In [6] it is explained that rather than having requests that may contain SQL injections being sent directly to a database, it will give a higher probability of preventing or taking action short after an attack if the system can give an output fast enough. The second part of the proposed system also includes the insertion of cross-site scripting detection before loading into the requested web page. Comparatively to the SQL injection detection's module position, Cross-site Scripting detection can be accomplished before the new page has loaded in. The reason remains the same as for the position of the SQL injection detection module, which is to prevent or take precautions shortly after Cross-site scripting has been attempted.

III. Technological Background

Kafka [20] is a distributed streaming platform that is highly scalable, fault-tolerant, and allows a high level of parallelism and decoupling between data producers and data consumers. Nowadays it is considered an industry standard when it comes to near real-time to real-time processing of streaming data. Meaning that Kafka is a critical component of most Big Data Platforms and hance the Hadoop ecosystem. Kafka consists of brokers which are Kafka nodes on a cluster and topics which are categories of streams and streaming records that can be partitioned and replicated and also support multiple writers (producers) and readers (consumers). Each broker has its own topic(s) and partition. The producer can push updated data into Kafka corresponding to its topic. In this case, the producer that produces messages and sends them to Kafka is the webserver and topics are GET, POST, PUT, and DELETE. The consumer can pull the updated data that the producer has pushed earlier from Kafka, which in this paper are the four Spark streaming jobs that detect intrusions.

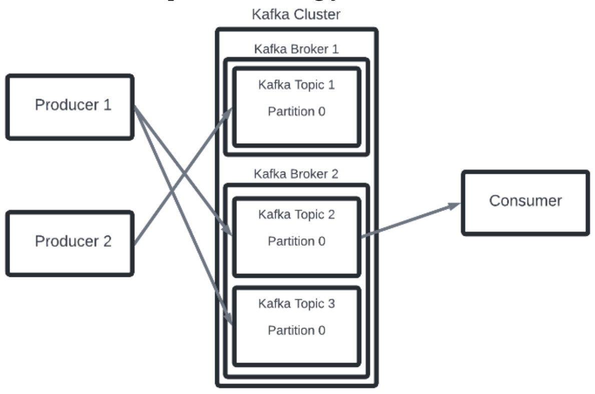


Figure 2 : Kafka Structure

Spark Streaming [19] is a framework used for large-scale stream processing. It achieves second scale latencies, that is the delay that should be minimized in seconds/milliseconds creating a near real-time to real-time processing time. Spark streaming requires a cluster to run spark jobs the same way Spark is running jobs. The major difference to Spark is that Spark streaming receives data in a stream formula, which in this paper's case is from a Kafka Topic. The received streaming data is split by a duration. The streaming data in each duration is then considered as a Batch, which afterwards can be used in Spark as a normal Batch. Batch operations of spark, such as Transformations that create new RDDs and Actions which return a result to the driver program are applied to RDDs in order to create the desired output.

Resilient distributed datasets (RDDs) are immutable and partitioned collections of objects or data. The immutability of RDDs makes these datasets unchangeable and hence does not create any synchronization problems from updating and overwriting. Instead, Spark creates a new RDD when a Transformation operation is used on an RDD. Additionally, the partitioning of RDDs allows the distributed computation when operations are called and guarantees no conflicts due to RDDs being immutable.

In this protocol IDS, four separate Spark Streaming Jobs are used of which all run on a Google Dataproc cluster [22]. The previously mentioned Jobs are Brute Force Job, HTTP Flood Job, SQLi Job, and XSS Job. Each of these jobs receives its streaming data from Kafka, which sends data to jobs according to their subscribed topic. This means that each Spark Streaming Job can connect to a Kafka broker and then choose to subscribe to one or multiple Kafka topics in that broker.

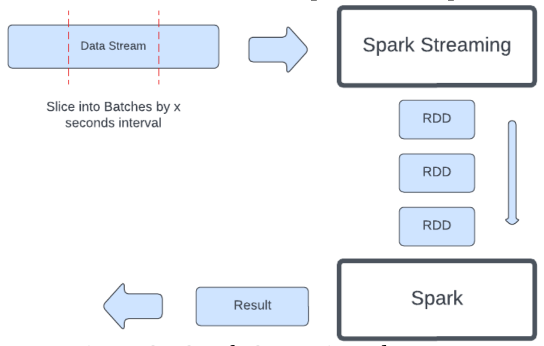
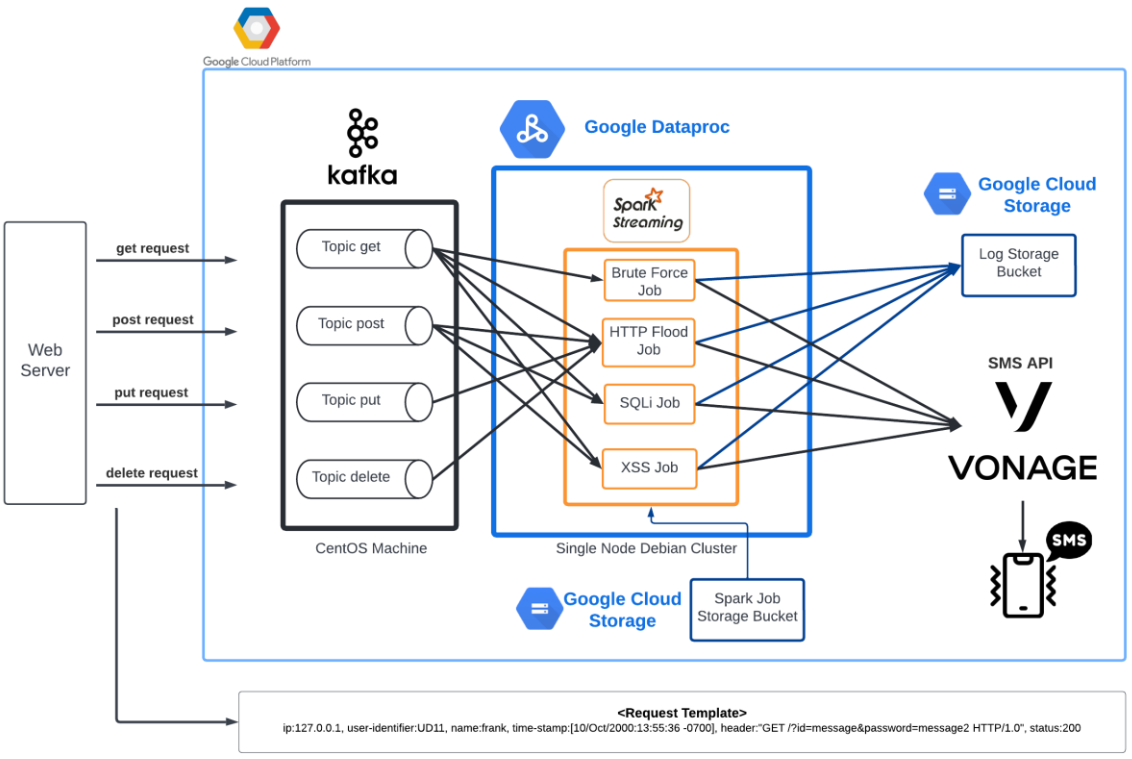


Figure 3 : Spark Streaming Flow



<Request Template>

```
ip:127.0.0.1, user-identifier:UD11, name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700], header:"GET /?id=message&password=message2 HTTP/1.0", status:200
```

Figure 4 : System Diagram

IV. Methodology

Figure 4 illustrates the overall system architecture and how the Protocol IDS system works. The system itself is entirely assembled on Google Cloud Platform [21], starting from HTTP requests which are directly received from the web server by Apache Kafka. Figure 4.1 shows a request template that will be received by Apache Kafka. This request pattern has been modified from Xu’s Visor system [15] log format since it contains all the important information needed in identifying intrusions. In addition, for future modifications, more information can be added to the HTTP request pattern depending on the web application’s vulnerabilities that depend on that system itself.

```
ip:127.0.0.1, user-identifier:UD11,
name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700],
header:"GET /?id=message&password=message2
HTTP/1.0", status:200
```

Figure 4.1 : Request Template

Apache Kafka then digests HTTP requests in the form of four topics: GET, POST, PUT, and DELETE, depending on the HTTP method type of each request. The HTTP request as a string is then sent to Spark Streaming, being the consumer of the topics, running on a Google Dataproc cluster [22]. Google Dataproc allows users to create and run jobs on different cluster types and versions. This project uses a single node cluster with the following versions installed: Debian 10, Hadoop 3.2, and Spark 3.1. A single node cluster in this case provides a single node that acts as both a master and a worker.

Internally Spark Streaming is divided and runs four isolated jobs: Automated Brute Forcing on web-based login job, HTTP flood attacks job, SQL Injections (SQLi) job, and Cross-Site Scripting (XSS) job. The output of each Spark Streaming job will be stored as a log file in a Google Cloud Storage bucket. In addition, we use the Vonage SMS API for SMS notifications [23], which will notify the security administrator if any of the algorithms detect an Intrusion attempt.

1) Automated Brute Forcing on web-based login detection Algorithm

With an Automated Brute Forcing on web-based login attemp, the system will detect exploits with the GET HTTP method, meaning that this Spark streaming job subscribes to Kafka’s GET topic. Using the flatmap method the request stream is split into different parts, where the argument being *\n* is used to split each line as its own data. Figure 4.2 shows the request string split by flatmap.

```
flatmap(_.value().split(“\n”))
```

```
(ip:127.0.0.1, user-identifier:UD11,
name:frank, time-stamp:[10/Oct/2000:13:55:36 -0700],
header:"GET /?id=message&password=message2
HTTP/1.0", status:200)

(ip:127.0.0.2, user-identifier:UD12,
name:john, time-stamp:[10/Oct/2000:13:56:01 -0700],
header:"GET /?id=message&password=message2
HTTP/1.0", status:200)
```

Figure 4.2 : Request string after using flatmap.

The algorithm counts the number of times an IP address calls the HTTP GET method in a particular time period. If the specified threshold is exceeded, it is considered a brute force attack and the algorithm saves the IP address and the number of attempts in a log file according to [5] [7]. Due to brute force login attempts varying in frequency and velocity as expressed in [8], where an attacker can use different hardware modifications to achieve an extreme number of attempts per second quota. But an amateur attacker might use more unsophisticated tools where the frequency of attempts may not be as high. Hence the threshold for determining what a brute force login attempt is, may not be the focus of this paper. Figure 4.3 shows examples of malicious IP addresses and the number of attempts for both Automated Brute Forcing on web-based login.

```
map(rdd => (rdd.split(“, ”)(0), 1)).reduceByKey((x, y)
=> x+y).filter(x => x._2 > requestThreshold)
```

```
(ip:127.0.0.1, 15)
(ip:199.12.26.22, 16)
(ip:149.244.168.229, 22)
(ip:42.40.79.231, 65)
(ip:219.57.228.72, 101)
(ip:3.111.203.205, 11)
(ip:29.13.130.4, 69)
(ip:39.119.21.208, 42)
```

Figure 4.3 : Examples of Automated Brute Forcing on web-based login and HTTP flood attack’s saved IP addresses and the number of attempts.

2) *HTTP flood detection Algorithm*

For the HTTP flood attack detection algorithm, the system will detect attempts of attacks through the GET, POST, PUT, and DELETE HTTP methods, meaning this algorithm is subscribed to every Kafka topic. Similar to the Automated Brute Forcing detection algorithm, first the requests streams are split into different parts. Secondly, the algorithm filters malicious IP addresses that have a number of attempts that exceed the limited threshold. Finally, the malicious attempts are saved into the log file saving location which is a Google Cloud Storage bucket. Figure 4.3 also shows the format of processed HTTP flood attacks.

3) *SQL Injection (SQLi) detection Algorithm*

With the SQL Injection (SQLi) detection algorithm, the system is attacked based on the GET and POST HTTP methods, so the algorithm subscribes and filters only those two Kafka topics. First, the algorithm must rearrange the request stream into a form of request strings that consists of the IP address as a Key and header part as a Value in the MapReduce architecture by using both the flatmap and map method. Figure 4.4 shows examples of request strings that have been rearranged by flatmap and map.

```
flatmap(_.value().split(“\n”)).map(x => (x.split(“, “)(0),
x.split(“, ”)(4)))
```

```
(ip:127.0.0.1, header:"GET /?id=admin" or
"1"="1&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message1&password=message2 HTTP/1.0")
(ip:29.13.130.4, header:"POST /?id=" or 1=1 –
&password=message2 HTTP/1.0")
(ip:149.244.168.229, header:"POST /?id= or 0=0 --
&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message5&password=message2 HTTP/1.0")
(ip:127.0.0.1, header:"GET
/?id=message6&password=message2 HTTP/1.0")
```

Figure 4.4 : Rearranged request string by flatmap and map methods.

The algorithm will filter the request strings that match SQLi request patterns which are found in the self-made list of hand-picked strings that are contained in most of the SQLi payloads from the dataset in [16] and save them into a log file. Figure 4.5 shows examples of request strings that match SQLi request patterns.

```
reduceByKey((x, y) => x + “ , “ + y).filter(x =>
sqli_payload_list.exists(y => x._2.contains(y)))
```

```
(ip:127.0.0.1, header:"GET /?id=admin" or
"1"="1&password=message2 HTTP/1.0", …. )
(ip:3.111.203.205, header:"GET /?id=' and
substring(password/text(),1,1)='7&password=message2
HTTP/1.0")
(ip:29.13.130.4, header:"POST /?id=" or 1=1 –
&password=message2 HTTP/1.0")
```

Figure 4.5 : Examples of request string match with SQLi pattern.

4) *Cross-Site Scripting (XSS) detection Algorithm*

For the Cross-Site Scripting (XSS) detection algorithm, the system will detect attempts of attacks via the GET and POST HTTP methods, hence this algorithm subscribes to and filters only those two Kafka topics. Similar to the SQL Injection detection algorithm, first, the algorithm rearranges the request streams into the specific pattern as in Figure 4.4 using the same method used prior in SQL Injection detection. After that, the algorithm filters malicious IP addresses that have a request pattern match with an XSS payload pattern in the self-made collection of commonly used strings that are part of XSS payloads in [17]. Finally, the algorithm saves all malicious attempts into a log file on the Google Cloud Storage bucket.

5) Request Generation Script

The Request Generation script has been created to generate massive amounts of requests for the system testing and evaluation phase. The script consists of Normal requests, Automated Brute Forcing requests, HTTP flooding requests, SQL Injection requests, and Cross-Site Scripting requests. One of the most important parts of the request is the header, which will further be used to classify whether the request is a normal request or a malicious request. The header of a normal request in this system will consist of a user ID and password. For the variety of requests, the algorithm will generate random passwords from the most popular password dataset list as in [18]. Automated Brute Forcing can be detected by a certain number of attempts in a period of time. Attacks can only be carried out with the GET command according to [5] and [7]. HTTP flood attacks are similar to the Automated Brute Forcing but can be attacked through all four Kafka topics (GET, POST, PUT, and DELETE). SQL Injections can be detected by malicious request header patterns which have been more elaborated on in [2] and [3]. The algorithm will randomly select an SQL Injection pattern from [16] and insert it inside the request header. Similar to SQL Injections, Cross-Site Scripting uses the pattern method as the detection method. The request header information will be randomly generated from the Cross-Site Scripting list which has been extracted from [17].

V. EXPERIMENTAL RESULTS

This paper's experimentation to acquire results includes the usage of the Request Generation script which is used for generating an immense number of requests to simulate real-world applications. Each Spark Streaming job is performed using four amounts of requests, being 10 000, 25 000, 50 000, and 100 000 requests respectively. The first value this experiment measures is the processing time of each Spark Streaming Job as can be seen in Figure 5.1.

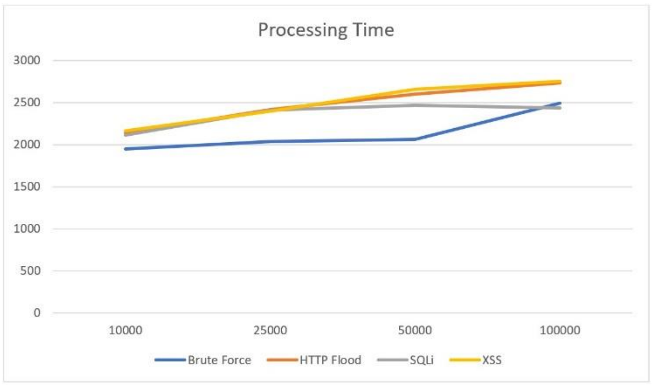


Figure 5.1 : Four Jobs Processing Time

The pattern of the plotted graph in Figure 5.1 goes hand-in-hand with Spark's advantage of being able to cache RDDs for efficient reuse and MapReduce's benefit of parallelization. The resulting graph shows that the processing time is between 2 and 2.8 seconds, where the number of requests is between 10 000 and 100 000. The time difference compared to the difference of the requests is minuscule to the point that it can be presumed that the rate of the processing time will increase in the same manner until the number of requests reached the limit of the computational power of the cluster.

Furthermore, we can determine the throughput of each Spark Streaming Job which we calculate as the value of Requests per Milliseconds. Once again, the results follow the theory of Spark which can be seen in Figure 5.2. As the number of Requests increases but the processing time remains close to the processing time of fewer requests, the throughput skyrockets. The graph shows that for all Spark Streaming Jobs the throughput builds up with the number of requests, hence proving that resource utilization is strikingly high when the amount of data that is processed is high, which correlates to real-world Web Applications.

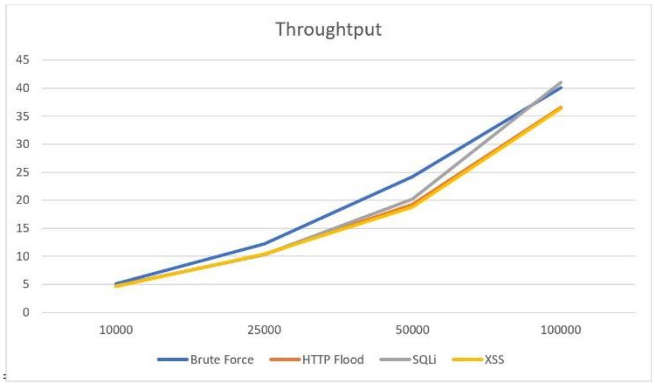


Figure 5.2 : Four Jobs Throughput.

Table 1 shows the detection confusion matrix for both the SQL injection detection algorithm and the Cross-Site scripting detection algorithm. The results convey the quality of the lists of patterns for both algorithms as the TP numbers for all cases are prominently high. For all of the request numbers, the percentage of the actual detected attacks compared to the total number of attacks averages at 89.38% for SQL injection and 97.5% for Cross-Site Scripting.

	TP	TN	FP	FN
10 000				
SQLi	86	9 888	12	14
XSS	96	9 900	0	4
Total	100 Attacks			
25 000				
SQLi	169	24 781	19	31
XSS	200	24 800	0	0
Total	200 Attacks			
50 000				
SQLi	260	49 694	6	40
XSS	299	49 699	1	1
Total	300 Attacks			
100 000				
SQLi	250	99 745	5	0
XSS	237	99 750	0	13
Total	250 Attacks			

Table 1 : SQLi and XSS detection confusion matrix for each request simulation.

Moreover, the number of false alarms, meaning that legit HTTP requests are being detected as attacks is relatively low, where there are on average 5 false alarms per 10 000 requests for SQL injection detection and 1 false alarm per
40 000 requests for Cross-Site Scripting. The reason for the big difference in false detection rate is the way SQL, which is used in SQL injections, is closer to Human Language than

JavaScript, HTML, and CSS, which are used in Cross-Site Scripting. Finally, the last number that is important being the FN rate, meaning attacks that are not being noticed or detected by the system, which is relatively low for all cases. On average SQL injections that are not being detected relate to 8 attacks per 10 000 requests and for Cross-Site Scripting the FN rate amounts to 5 attacks per 10 000 requests. These numbers can be reduced by adding additional strings to both lists that are used to compare requests for attacks, as currently, the system detects more well-known patterns. This correlates with this Protocol IDS being a Signature/Heuristic-based IDS, where one big disadvantage is that the system cannot detect intrusions that are not described in the signature collection.

VI. Final remarks

In this paper, a protocol IDS hosted on a cloud environment has been proposed, built, and tested for real-world usage. The background information for the architecture has been explored in-depth, which included Apache Kafka and Spark Streaming, how they work, and are implemented in this paper's proposed system design. The main intrusions the proposed system detects include Brute-force login attacks, HTTP flooding attacks, SQL injections, and Cross-site scripting. Each of the attacks' patterns have been explored and algorithms to detect them in Spark Streaming have been developed in order to match each attack's patterns, thus making this IDS a Signature/Heuristic-based IDS. The mentioned Spark Streaming algorithms have then been implemented on Google Cloud Platform alongside the Kafka cluster and the Storage space structure to both save the Jar files and log files of attacks that have been detected. To test and evaluate the system, multiple and varying sizes of requests have been created using a python script that matches the system's request pattern. Each Spark streaming job has been tested using 10 000, 25 000, 50 000, and 100 000 requests per batch. Not only are these numbers used to simulate the efficiency of the Spark Architecture, but also for the finding the detection rate of both the SQL injection detection algorithm and the Cross-site scripting detection algorithm, as they use dedicated pattern lists to detect strings in the requests. After running each Spark Streaming job, each with the four requests batch sizes, the evaluation has been done to show both the processing and throughput of the system. Both values for all Spark Streaming jobs show promising results of being able to handle up 100 000 requests per time frame without any problems. And for the confusion matrix, the detection rate, at this stage is acceptable, considering the damage intrusions such as SQL injection and Cross-Site Scripting can do to the system. To conclude, it can be said that the proposed system shows promising results of detecting intrusion in a real-time manner, while not taking up a lot of resources, have a somewhat low false positive and false negative rate, have high to permanent uptime according to the Cloud providers SLA, and cover a wide area of intrusion methods that it can detect.

References

[1] T. Kakarla, A. Mairaj and A. Y. Javaid, "A Real-World Password Cracking Demonstration Using Open Source Tools for Instructional Use," 2018 IEEE International Conference on Electro/Information Technology (EIT), 2018, pp. 0387-0391, doi: 10.1109/EIT.2018.8500257.

[2] C. Ping, "A second-order SQL injection detection method," 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), 2017, pp. 1792-1796, doi: 10.1109/ITNEC.2017.8285104.

[3] J. K. R, S. Balaji B, N. Pandey, P. Beriwal and A. Amarajan, "An Efficient SQL Injection Detection System Using Deep Learning," 2021 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE), 2021, pp. 442-445, doi: 10.1109/ICCIKE51210.2021.9410674.

[4] T. H. Hai and N. T. Khiem, "Architecture for IDS Log Processing using Spark Streaming," 2020 International Conference on Electrical, Communication, and Computer Engineering (ICECCE), 2020, pp. 1-5, doi: 10.1109/ICECCE49384.2020.9179188.

[5] L. Bošnjak, J. Sreš and B. Brumen, "Brute-force and dictionary attack on hashed real-world passwords," 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 1161-1166, doi: 10.23919/MIPRO.2018.8400211.

[6] P. N. Joshi, N. Ravishankar, M. B. Raju and N. C. Ravi, "Contemplating Security of Http From SQL Injection and Cross Script," 2017 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC), 2017, pp. 1-5, doi: 10.1109/ICCIC.2017.8524376.

[7] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert and C. Kemp, "Detection of SSH Brute Force Attacks Using Aggregated Netflow Data," 2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA), 2015, pp. 283-288, doi: 10.1109/ICMLA.2015.20.

[8] Keonwoo Kim, "Distributed password cracking on GPU nodes," 2012 7th International Conference on Computing and Convergence Technology (ICCCT), 2012, pp. 647-650.

[9] H. S. Shreenidhi, S. Prabakar and P. A. Kumar, "Intrution detection system Using IoT device for safety and security," 2021 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE), 2021, pp. 340-344, doi: 10.1109/ICCIKE51210.2021.9410730.

[10] B. Debnath et al., "LogLens: A Real-Time Log Analysis System," 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018, pp. 1052-1062, doi: 10.1109/ICDCS.2018.00105.

[11] S. V. Siva reddy and S. Saravanan, "Performance Evaluation of Classification Algorithms in the Design of Apache Spark based Intrusion Detection System," 2020 5th International Conference on Communication and Electronics Systems (ICCES), 2020, pp. 443-447, doi: 10.1109/ICCES48766.2020.9138066.

[12] M. T. Tun, D. E. Nyaung and M. P. Phyu, "Performance Evaluation of Intrusion Detection Streaming Transactions Using Apache Kafka and Spark Streaming," 2019 International Conference on Advanced Information Technologies (ICAIT), 2019, pp. 25-30, doi: 10.1109/AITC.2019.8920960.

[13] S. Honda, Y. Unno, K. Maruhashi, M. Takenaka and S. Torii, "TOPASE: Detection of brute force attacks used disciplined IPs from IDS log," 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2015, pp. 1361-1364, doi: 10.1109/INM.2015.7140496.

[14] Elmasry, Wisam, Akbulut, Akhan and Zaim, Abdul Halim. "A Design of an Integrated Cloud-based Intrusion Detection System with Third Party Cloud Service" Open Computer Science, vol. 11, no. 1, 2021, pp. 365-379. https://doi.org/10.1515/comp-2020-0214

[15] Wenyi Xu (2017) Visor: Real-time Log Monitor [Source Code]. https://github.com/xuwenyihust/Visor

[16] payloadbox (2021) SQL Injection Payload List [Dataset]. https://github.com/payloadbox/sql-injection-payload-list

[17] payloadbox (2021) Cross Site Scripting (XSS) Vulnerability Payload List [Dataset]. https://github.com/payloadbox/xss-payload-list

[18] iryndin (2018) 10K-Most-Popular-Passwords [Dataset]. https://github.com/iryndin/10K-Most-Popular-Passwords

[19] https://spark.apache.org/docs/latest/

[20] https://kafka.apache.org/documentation/

[21] https://cloud.google.com/

[22] https://cloud.google.com/dataproc/

[23] https://www.vonage.com/communications-apis/