Implementation of a Web Application Firewall for a High Availability front end

Hernan Espinosa Reboredo

Abstract— Nowadays, thanks to globalisation, companies and their bussines are growing fast, making their main objectives to keep up with market demands to satisfy user needs. Since market demands are not static, high availability is a requirement that a company infastructure must meet. If any company wants to stay competitive in these circumstances, it must be able to change what it is doing to fill the needs and wants that customers have. Therefore, companies experience growth in its resources, services and data while still having to provide the efficiency and the quality of service of the offered services. This study aims to determine how companies should protect their services and how they could scale them by analysing, designing, implementing and testing a high availability web application firewall in order to avoid security threats and, as the company's systems grows, to continue meeting market demands efficiently and securely.

Index Terms—Web application firewall, web application, load testing, server security, HAProxy.

Resumen— Actualmente, gracias a la globalización, las empresas y sus negocios están creciendo rápidamente, lo que hace que sus objetivos principales sean de satisfacer las demandas del mercado y satisfacer las necesidades de los usuarios. Dado que las exigencias del mercado no son estáticas, la alta disponibilidad es un requisito que deben cumplir las infraestructuras de la empresa. Si alguna empresa quiere mantenerse competitiva en estas circunstancias, debe ser capaz de cambiar el enfoque de lo que está haciendo para satisfacer las necesidades y las necesidades que los clientes tienen al momento. Debido a ello, las empresas deben proporcionar la eficiencia y la calidad del servicio de los servicios, aunque experimenten un crecimiento en sus recursos, servicios y datos. Este estudio tiene como objetivo determinar cómo las empresas deben proteger sus servicios y cómo pueden escalarlos analizando, diseñando, implementando y probando un web application firewall de alta disponibilidad para evitar amenazas de seguridad y a medida que crezcan los sistemas de la empresa, poder continuar satisfaciendo las exigencias del mercado de manera eficiente y de forma segura.

Palabras clave—Firewall de Aplicación Web, aplicación web, test de rendimiento, seguridad de servidores, HAProxy.

1 Introduction

THE aim of this project is to build a high availability web application firewall (WAF) [1] that will monitor and analyse client's requests data, keeping in mind the protection of backend server's security.

A WAF helps to protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. By deploying a WAF in front of a web application, a filter is placed between the web application and the Internet, increasing security, performance and reliability by having client's requests pass through the WAF before reaching the servers. A WAF operates through a set of rules often called policies. These policies aim to protect against vulnerabilities in the application by monitoring and filtering out malicious traffic.

In this project, we are going to build a WAF for a high availability front end, and we are going to carry out a comprehensive analysis of the events that take place in the built architecture, using development tools to monitor the traffic.

The main reason that has motivated this work is to improve the security of servers, to have a better control over the events of their activity, and to reduce the time of action after an incident. Therefore, this project is appliable to any web application architecture in order to improve its security.

2 State of the art

The WAF that we are going to build is going to be deployed to a high availability front end and a lot of work has been conducted in this field, providing many available options in order to suit different use cases and needs [2][3]. The increasing use of web applications, the wide variety of platforms that need to run on and the flexibility and speed of today's applications, have made companies offer cloud based WAF [4][5][6] to integrate security solutions into the applications and apply external security. The cloud Based WAF continuously monitors and protects the servers from incidents, all offered in a package which provides quick response against threats. The cloud based WAF's offer its

- *E-mail of contact: hespinosar@gmail.com.*
- *Degree Specialisation taken: Technologies of Information.*
- *Work supervised by: Sergi Robles Martínez (DEIC).*
- *Year 2020/21.*

services to the applications in real-time using AI-enhanced detection methods, behavioural analytics, application-layer encryption and databases. These tools are used for securing certain functions and trasactions as part of an applications internal process, preventing vulnerabilities and zero-day threats.

While cloud based WAF's play a critical role in securing the web applications, it makes it a challenge to deploy a cloud based WAF to a web application, since there is a necessary expertise required to deploy and manage traditional WAF's and its software and hardware must be maintained in order to work correctly. Companies like Barracuda [7], Amazon [8], FortiNet, [9] or Altair [10], are offering WAF-as-a-Service to simplify the deployment of security to an application. A WAF-as-a-service can be deployed in minutes with initial configuration and the client developers won't have to maintain any hardware or software focusing their time dealing with the critical aspects of the application's security.

3 Objectives

The WAF developed in this project must be able to protect web applications by filtering and monitoring HTTP traffic between a web application and the Internet. We have planned a set of objectives to achieve by the end of this work. The objectives are listed in incremental priority:

1. To learn how a WAF works, and how it is used.
2. To identify which software tools, we are going to use to develop the project.
3. To learn how the different software tools work, and how they are going to be for use, to make the project possible.
4. To design, implement and test a scalable high available client-server web application architecture.
5. To apply the data persistency and load balancing feature to the web application firewall for redundancy and performance efficiency purposes.
6. To develop and add rules to the service to improve security.
7. To do the log management to analyse all the client requests that are done in real-time as well as to establish rules to grant or deny access to the backend servers.

4 Methodology and Planning

This project follows a software development model based in incremental prototyping [11]. Each prototype will fulfil a set of requirements. The final prototype will accomplish the objectives from this work. Each prototype built will verify the definition of requirements, since requirements can change from the initial requirements definition.

This work is divided into three main phases: Web Application Firewall version 1 (WAF_v1), Web Application Firewall version 2 (WAF_v2) and Web Application Firewall version 3 (WAF_v3). For every phase these steps are going to be considered for the development: The architecture analysis, to justify the proposed structure, explaining the decisions made and reasons why the structure is suitable; The architecture design, to design the proposed structure, explaining the decisions made and reasons why the structure is suitable; The architecture implementation, to implement the design proposed, and Functional testing, to validate that every project phase satisfies its initial requirements. For the final phase of the project, different types of tests are going to be performed to ensure that it can perform correctly under high load.

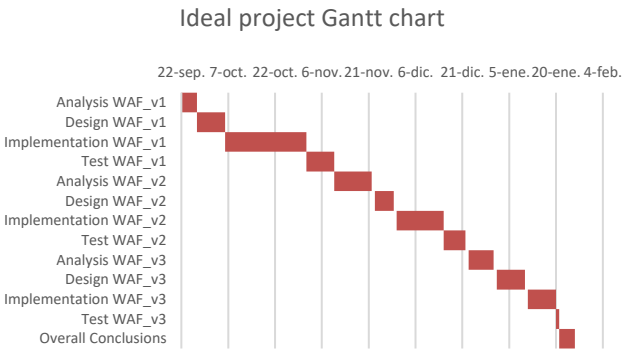

Fig. 1. Shows the ideal Gantt chart diagram for this work.

To keep track of the development of this project, we are going to use the Gantt chart, which is a useful way of scheduling a project and defining the different dependencies between tasks. In Fig. 1, we can see the layout of the ideal progress from the phases of the project throughout time. This project is limited by time and budgetary resources. The main developer leading the project fits in the salary category of an undergraduate engineering student, in the Universitat Autonoma de Barcelona. The salary for a category 2 undergraduate engineer is 18,44 €/hour, in conformity to the agreement for support research staff of the University, which can be found at the univeristy's website. The development includes 280 hours dedicated to the development of the project, which are held accountable at the salary, and 20 hours dedicated to meetings with the project supervisor and research staff, and the paper writing, which won't be held accountable for the salary. The project is limited by the salary of the developer and the time limit of 300 hours. To conduct the project, the budget is 280€ x 18,44 €/hour, 5163€.

5 Tools for the Development

In this section of the paper we are going to describe the set of software that is going to be used to develop this work. This project will be developed using a Linux operating system, Ubuntu.

5.1 Docker: The development environment

Docker [12] is a tool designed to make it easier to create, deploy, and run applications by using containers [13]. Containerization allows us to package up an application with all the parts it needs, such as libraries and other dependencies, and run it all out as one package. Thanks to

the ease of use of the Docker environment management, it makes it a suitable environment for this project. Separating the different components of our web application service into different containers will have security advatages, because if one container is compromised, the others remain unaffected. Separating the different components of our web application into different containers will avoid conflicts with dependencies, therefore, gaining practicability.

5.2 HAProxy: building tool

For the development of the loadbalancers, we are going to use the HAProxy software because its grately used for developing high available load balancers and because its open source. HAProxy is a very fast and reliable solution offering high availability, load balancing and proxying for TCP and HTTP-based applications. It is particularly suited for web application services.

5.3 Keepalived: building tool

Keepalived is a routing software which provides facilities for load balanceing and high availability. The software provides transport layer loadbalancing implementing a set of checkers to dynamically and adaptively maintain and manage loadbalanced server pool according their health. High availability is achieved by Virtual Redundancy Routing Protocol (VRRP) [14].

5.4 Elastic Stack: Data Management tool

The Elastic Stack [15] is a powerful search engine which will allow us to process logs generated from the architecture. Elastic Stack is a complete end-to-end log analysis solution which helps in deep searching, analysing and visualizing the logs generated from different machines.
Elastic stack will help us search through the multiple logs at a single place and identify the issues spanning through multiple servers by correlating their logs within a specific time frame found in our environment.

5.5 Apache AB: Testing tool

Apache AB [16] is an open source testing tool developed by the Apache organisation used for benchmarking an HTTP web server. Apache AB will be used to test and measure the performance of the built architecture.

5.6 Github: Version Control tool

GitHub is a versioning tool which will be used, in order to keep copies and record of the work done over the course of this project. A repository has been created in GitHub [17] where the project is updated, and version controlled.

6 Phase 1: WAF_v1

6.1 Analysis

The first prototype of the project consists in building a client-server architecture composed of three nodes. The client node, which generates the requests, the intermediary node, which will be used to secure the server node, and the server node, which will handle the client requests and will generate responses. The architecture will serve an index.html file stored in the backend server. The objective of this phase is to develop and simulate a web server architecture including the intermediary node, which will work as a load balancer in future prototypes. This phase will work as a base for future upgrades in the next phases. Different resources and processes will be carried out to suit the different objectives from the work. With this phase we want to improve backend server's security and reduce security vulnerabilities from the web application service.

To keep up with the main objectives from this work, we are going to develop data persistence in the nodes which serve the client requests. For this phase of the project, we are going to manage architecture networking and we are going to centralise the logs from the service's activity.

6.2 Design

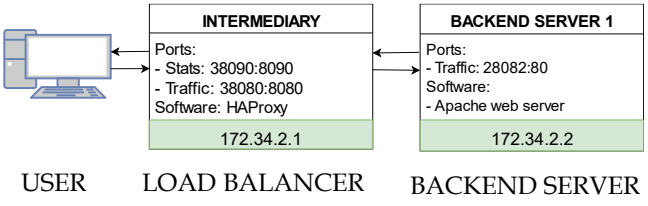

Fig. 2. Shows the UML component diagram from the design structure proposed for phase one of this work.

In Fig. 2, we can observe the layout of the three nodes which compose WAF_v1. Both Intermediary and Backend Server 1 nodes are built on Docker containers and a private network has been designed to host the architecture. The Backend Server 1 node consists of an Apache web server which serves an index.html file that will be requested by the user client.
In this phase of the project, the Intermediary node of the architecture is built to hide the backend server node's IP from the Internet, making it more secure. Every time a user requests the index.html file from the backend server, instead of connecting to the server endpoint directly, the user will request the file using the intermediary's endpoint, which works as a proxy. The intermediary node will redirect the client's request to the server as well as it will redirect the server's response to the user. Because of the need of transparency and privacy of the backend server, we have opted to use the HAProxy software in the intermediary node, as its software can be configurable to work how we want to. Since we want to be able to run the Docker containers always with the same configuration, data persistence has been developed. Since we want to monitor the activity from our architecture, logging has been configurated so we can track requests and verify that everything works correctly.

6.3 Implementation

Since we are using containerization to develop our work, data does not persist when a container no longer exists, and it can be difficult to get the data out of the container if another process needs it. To manage this problem, we are using Host-based Persistence [18] among containers to store files in the host machine, even after the container stops. By using Host-based persistence, data presits

outside of the container, which means it will be available even after a container is removed. In WAF_v1, data persistence is important since we need our architecture's containers to centrally store logs to the same directory, making it easier to process the logs, so we can test the architecture's performance. For log management, we are going to use the software Rsyslog. Rsyslog is a powerful, secure and high-performance log processing system which accepts data from different types of source and outputs it into multiple formats. Redirecting all the logs from rsyslog to the standard out device makes logs be compatible with docker default logging. In this configuration, the Intermediary node uses the software HAProxy to load balance traffic across the architecture. Regarding HAProxy event logging, the software, does not log to stdout by default. To solve that problem, we built a configuration which takes the logs generated by the software and sends them to a specific local directory both in the container and in the local machine, where the containers are executed.

Since the design is scalable, in a future situation we may have the use case where each backend server is connected to different databases to serve client requests, or we may have the situation where we have multiple load balancers that are interconected. Data is generally replicated to enhance reliability or to improve performance. One major problem is to keep replicas consistent. Data base servers as well as the load balancers must be configured to perform master-master replication as load balancing involves both reading and writing to all backends.

When configuring the network for the architecture environment in this phase of the project, a private network has been used to develop the infrastructure. The HAProxy intermediary node, is a basic load balancing node which listens on a specific IP address and port, then forwards the incoming traffic to a specified server.

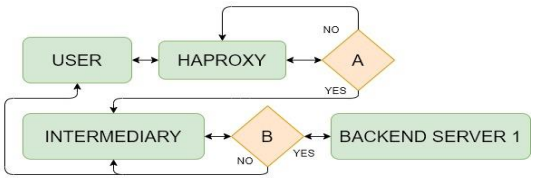

Fig. 3. Shows the UML flow diagram from WAF_v1 of this work.

In Fig. 3, there are represented in incremental order of priority, from letter A to B, the different possibilities from which the route of the request can take, through the architecture, in flow of execution. In this use case, when the architecture is set up, the intermediary is the load balancer node, and the backend server 1 is the backend server which serves the index.html file.
By typing to any web browser, the following URL address, 172.43.1.1:28080/index.html, composed by the endpoint from the intermediary node, the index.html file stored in the Apache backend server node is requested by the user.
Possibility A takes place: If the intermediary node is available, the request will be handled by the node in question. The HAProxy software will handle the request and will forward it to the backend server. If the intermediary node is not available, the HAProxy software will send to the user an HTTP 503 error response advertising the unavailability of the service. Then, possibility B takes place: If the backend server is available, it will deal with the response, which will be delivered to the client by the intermediary node. If the backend server is not available, the intermediary node will send the user an HTTP 503 error response advertising the unavailability of the service. Note that in any of the events, the backend server IP has been exposed to the Internet, but the intermediary node IP has been exposed.

Thanks to data persistency, and the software Rsyslog, the logs from the http request will be stored in a directory from the host's computer and we are able to trace the client request and response.

6.4 Test

In this part we are going to perform a functional implementation test, to the architecture to verify that the prototype follows the initial requirements.
Since the main objective of this phase is to build a base client-server architecture, functional testing is going to be performed for quality assurance. Black-box testing will be performed, which bases its test cases on the specifications of the software component under test. Functions are going to be tested by feeding them input and examining the output. In this phase, stress tests and other testing, are not going to be performed.

For the test, we are going to simulate a test case where a single user performs a single HTTP request to the index.html file. The web application performance tool that we are going to use is ApacheAB. The tool is designed to give a impression of how the current Apache installation performs. The aim of this test is not to test how many requests per second our Apache installation can serve, but to ensure the identification of functions that the software is supposed to do.
By opening a new terminal window in the virtual machine environment and typing the following command line: ab -n 1 http://172.34.2.1:8080/index.html we are performing a functional test which the index.html file is requested one time. The test will finish when the ApacheAB software receives a response from the server. The tool gives us information about the request, such as the server hostname, IP, port and the document requested. Since the Intermediary node is the main load balancer from the architecture, by checking the log from the request, we can assure that the request has gone through the path expected. By analysing the log file from the HAProxy software from the intermediary node and the results from the ApacheAB test, the request follows the path designed and the architecture performs as expected, allowing us to start the WAF_v2 development.

7 PHASE 2: WAF_V2

7.1 Analysis

In this phase of the project, WAF_v1 has been upgraded by developing high availability load balancing and data redundancy to the architecture, by adding two load balancers in the front-end of the architecture. The main objective of this phase of the project is to set up a simple IP failover between two servers, which will be the load balancers. We want to develop and understand how high availability load balancing works, how different configurations can have an impact to the architecture's performance and usability, and to improve backend server's security.

In order to carry out the objective, there has been an improvement regarding the architectural configuration from the WAF_v1's set-up, by the addition of more nodes either on the frontend side of the architecture or on the backend, and the improvement in the software configuration that will suit the architecture's needs.

Functional tests have been performed, as well as the comparison between WAF_v1 and WAF_v2 from the project, in order to analyse the importance of a high availability load balancing set up. Different resources and processes have been carried out to suit the different objectives from the work.

7.2 Design

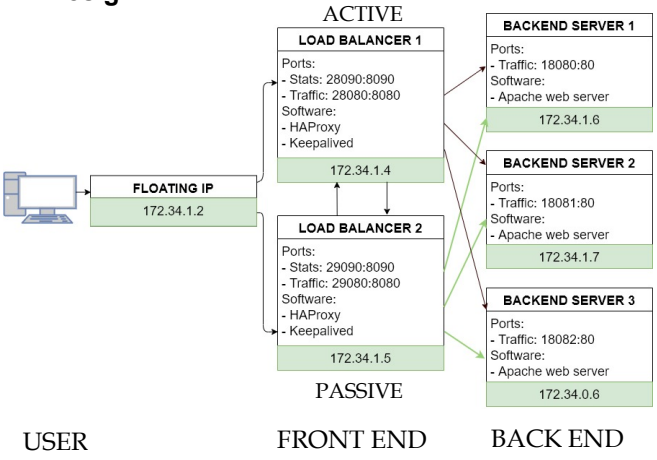

Fig. 4. Shows the UML component diagram from the design structure proposed for phase two of this work.

In Fig. 4, we can see the layout of the nodes which composes the architecture's prototype. Every node in this configuration is built on Docker containers and a private network has been designed to host the architecture. Backend server one, two and three consist of an Apache web server which serves an index.php file, containing the path of the request throughout every node. In this version of the WAF we have opted to use the index.php file since PHP has multiple native methods from its language, which makes it easier to get the IP and Hostname from a certain node from the architecture, and to display it in the file. The file is the element which will be requested by the user client. In this phase of the project, the load balancers are built to provide robustness to the architecture. Every time a user requests the index.php file from the backend servers, instead of connecting to the server endpoint directly, the user will request the file using the intermediary's endpoint, a virtual IP, which works as a proxy which will redirect the request to the load balancers. Considering high availability, one of the load balancers will be an active node, which will redirect the request to the backend servers and will redirect the response to the User. The other load balancer will be a passive node. If the main server fails or it's unavailable, the passive node, or the one that is available, takes the master role and can redirect and balance client requests throughout the architecture, making the design more secure and reliable. Because of the need of high availability as well as keeping transparency and privacy of the backend servers, we have opted to use Keepalived software alongside HAProxy software to run the load balancers, as its software has been configured to work according to the requirements of the project. The backend of the architecture consists of three Apache web servers, which will respond the index.php file that will be requested by the user, to test that the front-end is working as expected.

7.3 Implementation

In this version of the project we have set up a two-node front end load balancer in an active/passive configuration with HAProxy and Keepalived. The load balancers sit between the user and the three backend Apache web servers that hold the same content. The backend servers from this configuration reassemble a company's service backend. Not only does the load balancers distribute the requests to the three backend Apache servers, but they also check their health to check their availability. If one of them is down, all requests will automatically be redirected to the remaining backend servers. In addition to that, the two load balancers monitor each other using the software Keepalived, and if the master fails, the slave becomes the master, which means the users will not notice any disruption of the service. Keepalived is the software that manages the configuration of the Virtual Floating IP (VIP) from the architecture frontend. Thanks to Keepalived and HAProxy, we can scale our architecture by adding more load balancer nodes at any time in the future if it is needed. Haproxy performs load balancing on layer 7, HTTP and TCP-based services. Keepalived uses VIP's to perform load balancing and failover tasks on the active and passive routers. All nodes running Keepalived use the VRRP protocol.

HAProxy lets configure the load balancer to load balance traffic to the backend servers using four different algorithms. A static Round Robin, where each backendserver is used in turns per their weights and it is the algorithm used for the configuration of the WAF in this work. Unlike the Round Robin algorithm, changing server weight during execution is not an option. When a server goes up, it is immediately introduced into the farm once the full map is recomputed and the load balancing algorithm does the scheduling. Other algorithms, like Least Connections, Source, URL or URL parameter can be used. More information can be found in the HAproxy configuration website. Depending on the situation and the use case of the architecture, an algorithm or another will be used. Using a

Least Connections algorithm is useful when very long sessions are expected, such as SQL, LDAP, etc. whilst using a Round Robin algorithm is useful when we want to assign equitable weight to every server from a net of servers.

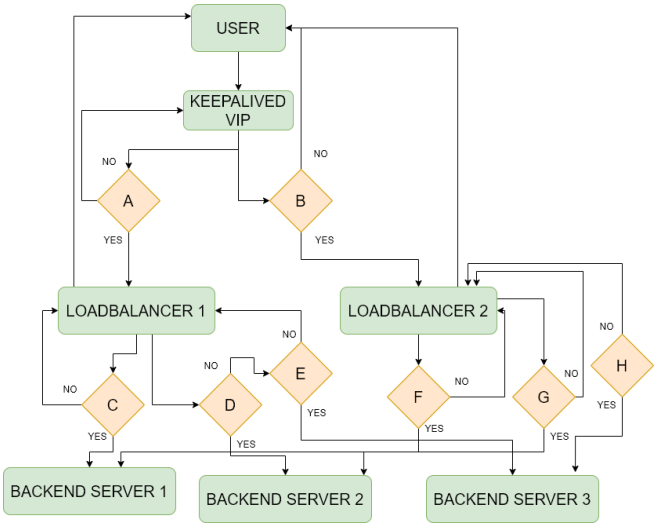

Fig. 5. Shows the UML flow diagram from WAF_v2.

Due to the complexity of WAF_v2 architecture, in Fig. 5, we have arranged the representation of a UML flow diagram reassembling an example use case from a user requesting the index.php file from the backend servers. In the figure, there are represented, in incremental order of priority, from letter A to F, the different possibilities from which the route of the request can take through the architecture's use case, in flow of execution. In this example, when the architecture is set up, the loadbalancer one is the master node, and the load balancer two is the backup node. The algorithm that the HAProxy is using to balance the traffic to the backend servers, in the current in the current configuration, is a RounRobin algorithm.

When the Users wants to acces the service, they type to any web browser the following URL address, 172.43.1.2:8080/index.php, which is the VIP endpoint. Through the VIP we are accessing through the virtual endpoint to the master node. Thanks to the Keepalived software, the software in the master node sends periodic advertisements to the other load balancer, to check its status of operation. Possibility A takes place: If Load balancer 1(LB1) is available, the request will be handled by the node in question. If LB1 is not available, possibility B takes place: If Load balancer 2 (LB2) is available, which is the backup node, the VRRP instance determines the running status of the active node. Since the active node fails to advertise after a previously configured interval, Keepalived initiates failover and its status will change to master. As for LB1, it will change its status to inactive. In LB2, the HAProxy software will handle the request and will forward it to the backend servers, performing layer 7 traffic load balancing. If LB2 is also not available, an HTTP 503 error response is given to the user, advertising the unavailability of the service. Concerning the backend servers, possibility C goes in pair with possibility F, but changing the backendserver node. The same happens with possibility D and G, and possibility E and H. Possibility C and F: If the backendserver is available, the request will be handled from the load balancer to the backendserver node. If the backend server is unavailable, since replication is implemented for the backend servers, then possibility D and G take place. If the other backendserver is available, the request will be handled by the node available. If either of the backend servers are not available, since replication is implemented for the backend servers, then possibility E and H take place. If the backendservers are not responding, the user will be informed with a HTTP 503 error response, generated by the load balancer, advertising the unavailability of the service. For testing purposes, to follow the trace that the request has taken, the index.php file shows the request's route taken, by showing the IP and Hostname where the request paquet has passed. As well as with the index.php file, by checking the load balancer logs, we can overview that the traffic from the client request and response goes through all phases anticipated, and we can corroborate that the implementation of the design works.

Thanks to the upgrades to WAF_v1, the internal implementation of our architecture is not exposed to the public internet. By having multiple nodes for both the front end and backend, front-end high availability has been developed, making the design more robust, fault tolerant and reliable to overtake large amounts of client requests.

7.4 Test

In this part we are going to perform an implementation test to the architecture to verify that the prototype meets the functional requirements. Since the main objective of this prototype is to perform high availability load balancing with the client requests, functional testing is going to be performed for quality assurance. In this test, functions are going to be tested by feeding them input and examining the output.

For the test, we are going to simulate a test case where multiple users perform queries requiring the index.php file. We are going to use ApacheAB, the same tool used for testing WAF_v1. The aim of this test is not to test how many requests per second our service can serve, but to ensure that the loadbalancers does serve the requests. The testing and analysis of how many requests per second our architecture is capable of serving under heavy load, the point of failure of our architecture, the average response time of the architecture under different loads and the maximum number of requests per second that the architecture can handle will be conducted in the performance testing section of this paper.
The test is performed with the Keepalived configuration with LB1 as the master node and LB2 as the backup node. The HAProxy node's software load balancing algorithm is set to Round Robin. The actual test is performed with a loadbalancer default configuration, meaning that there is not rate limiting or SSL termination. Since the backend

from the architecture is composed by three backend servers, and the algorithm used to distribute the load is RoundRobin, we are going to perform 3 queries, so we can assure that all backend servers are requested the index.php file.

By opening a new terminal window in the virtual machine environment and typing the following command line: ab -n 3 http://172.34.1.2:8080/index.php, we are performing a functional test where the index.php file is requested three times. The test will finish when the ApacheAB software receives a response from the servers. By checking the log from the requests, in the master node, LB1, we can see that the requests have followed the expected path.
Every time a request is made, if the LB1 stays as master node, the request will follow the path from the client IP to the backend server three's IP, all the way through the LB1. If the load balancer's status changes, and the second load balancer is available, the path from the queries will change and the request will be handled by the LB2. If any load balancer is added to the architecture, its status would be backup, meaning that in case of failover, the nodes would be able to take the task of a master node.

The request conducted is simple, therefore, the weight in processing power is very little. By analysing the log file from the HAProxy software from the loadbalancers, and the results from the ApacheAB test, the request follows the path designed and the architecture's loadbalancers forward requests, passing the test performed.

8 Performance testing

There are two perspectives when testing a WAF. The first perspective is to test the functionality of the set of software components implementing the functional requirements. In this work, each prototype's functional requirements have been tested in the previous sections. The second perspective when testing a WAF, is to test that the running environment composed of the hardware, software and middleware components needed to execute the WAF, must meet the requirements to provide the service to a user. When testing the WAF, it is important to take in consideration both perspectives, since they are complementary. Also, when testing the WAF, it is important to take into consideration that the aim of Web application testing consists of executing the application using combinations of input and state to reveal failures. A failure is the manifested inability of a system or component to perform a required function within specified performance requirements. Therefore, in this part of the work different types of testing will be performed in order to find out the point of failure of the WAF.

8.1 Functional Testing

A WAF can be considered a distribuited system, wth a client-server architecture, including the users that are accessing the system, which could be distribuited all over the world. The heterogeneous nature of hardware and software which compound the system, and the ability of the system to generate software components at run-time, depending of input of the client's requests, makes it important to test the WAF's functionality. In order to verify the system requirements of the project, testing is required to find failures in the service/functionality in order to test that the behavior of the application is consistent with the service requirements. For each phase of the work, unity testing, has been conducted to the WAF, where the type of unit that has been tested is the front-end web page. Integrity testing has also been conducted, where the interaction of the different parts which compound the software have been tested together as a whole, identifying failures due to their coupling. System testing has been tested to test the behaviour of the web application, where the navigation throughout the web application has been tested as well as the web application responses from client requests, which in every case, there were not any web application failures due to incorrect redirection of web pages.

8.2 Non functional Tesing

While testing the WAF, there are different non-functional requirements that a web application should satisfy, such as performance, scalability, compatibility, accessibility, usability, and security. For verifying each non functional require-ment, testing activities with specifc aims will have to be designed.

To evaluate performance, we have tested server response time and the service availability. We have tested 13 different simulations of regular user activity, the first test being one user requesting the service to the WAF and the 13th test being the test where the service reaches the point of failure. For every test we have monitored the web application response times involving simultaneous users accessing the WAF. Measurements have been taken from the loadbalancer one of the architecture, since in this work, loadbalancer one is set to be the active node in the WAF's configuration. To verify the response time, we are expecting a response time of no more than 1 second. WAF_v2 has been designed so it can easly be horitzontally scaled, by adding more loadbalancers, or vertically scaled, by adding more power resources to the loadbalancers to cope with the web application requirements, so it does not become a bottleneck between the user and the web application. To monitor the system testing, the log files have been monitored and analised.

The WAF_v2 is running on Docker containers, each using 4MBytes of RAM, that are running on a Linux Ubuntu VM powered with 8Gb of RAM, and a single core of an Intel I7700HQ processor. By default, Docker containers are designed to use as many power resources as the host's kernel scheduler allows. Memorywise, in order to not encounter situations where the system could get an out of memory exception, the docker daemon attempts to mitigate the risks of such failure. If the value of the memory flag on the container setting is not specified, the minimum amount of memory that the container will use is 4Mbytes, as the present configuration on our containers. CPUwise, we can configure the Linux Kernel CPU scheduler to configure the amount of access to the CPU resources our containers

have. All containers use as much CPU as they need. The more resources the containers have, the better they will perform. Since we are running several programs within the same Ubuntu machine, we might consume more CPU, which can affect other programs while dealing with parallel runs, dissortiing our results. Docker relies on the amount of HW resources given, it does not resolve HW scaling problems, therefore the host OS and the container configurations need to be tuned. The size of the file requested to the WAF is 437 Bytes.

The unity that it is used to test the WAF performance is the number of responses over a period of time, how many requests can the architecture handle before failing. That parameter will be given by the software testing tool Apache AB, for every measurement taken. The type of load testing that we have conducted to our WAF is steady load testing. Steady load testing, where a specified number of virtual users' requests are sent to the architecture at once. We have conducted different measurements by increasing the number of parallel virtual user requests, from one, up to the breaking point of the service. Each test will be done taking in consideration two parameters: Parameter N, which represents the number of requests to perform for the benchmarking session; and parameter C, which represents the numer of multiple requests to perform at a time by a virtual user. By increasing the number of virtual users, different parameters from the loadbalancers and the backend servers, like memory and CPU utilisation are monitored. We have obtained the following results:

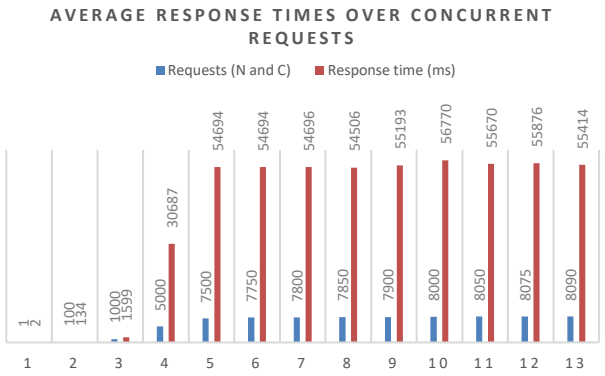

Fig. 6. Shows the graph showing the load balancer one, average response time for each number of virtual users generating concurrent requests during the 13 measurements of the test.

Requests (N and C)	Max. CPU Usage (%)	Max. Memory Usage (%)
1	0,5	0,11
100	2,5	0,12
1000	19	0,25
5000	24	0,4
7500	22	0,4
7750	11	0,42
7800	17	0,36
7850	18	0,44
7900	17	0,44
8000	17	0,39
8050	16	0,39
8075	18	0,37
8090	22	0,45

Fig. 7. Shows the table displaying the loadbalancer one average CPU and memory utilization during 13 tests.

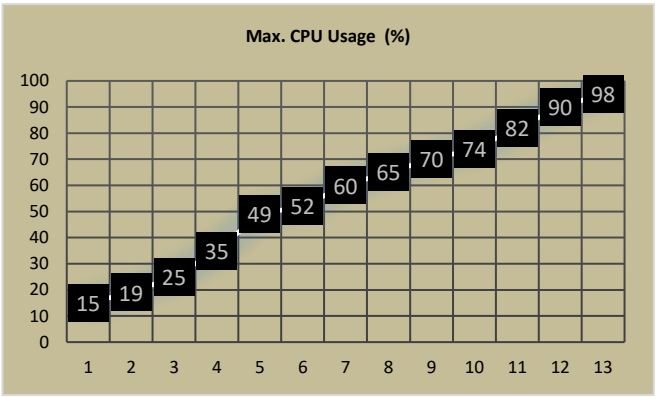

Fig. 8. Shows the table displaying the backend server's average CPU utilization during the 13 measurements of the test.

In Fig. 6. the graph represents the average response times obtained from load balancer one, the first test with parameters N = 1 and C=1, and the last test with N = 8090 and C = 8090 concurrent virtual users. The graph shows that response time increases as the load is increased, as expected. Fig. 7, shows the CPU and memory utilisation from the loadbalancer one, while during the tests. It is observed that during the test, as the number of concurrent virtual users increases, the CPU and memory usage from the loadbalancers increases as well, but not as we expected as if the loadbalancer would become oversaturated, that we would be expecting a 100% of CPU usage. There is a peak of 22% of the loadbalancer's CPU usage that has been monitored at the 13th test, before system failure, at N = 8090 and C = 8090 parameter measurement. Observing Fig. 9, we can see that the loadbalancer's requests/second times drops as the N and C parameters keep increasing, till the breaking point of the service, fixed at N = 8090 and C = 8090. This situation makes us consider cheking the backend servers benchmarks, to consider if the backend servers are being the bottleneck of the service.

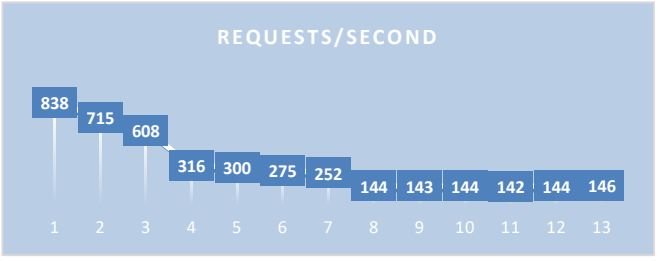

Fig. 9. Shows the graph displaying the load balancer one request/time rate, during the 13 measurements of the test.

The system performs as expected till we hit N = 8090 total users and C = 8090 concurrent requests during the test. After that measurement, the system is stressed beyond its specification limits, the service crashes and needs to be restarted. The part of the architecture that is failing, are not the loadbalancers, but the backendservers. Observing Fig. 9, which represents de monitorisation of the different

measurements of requests over time of the loadbalancer, and checking Fig. 8, the average CPU activity of the backendservers, we can see that the backend server's lack of multithreading capabilities and memory limitation to cope with that amount of load, brings the WAF capabilities to handle 838 requests/second at the first test, using 15% of backend server's CPU average usage, to 146 requests/second, at the last measurement, using 98% of backend server's CPU usage. After the last measurement, the apache servers oversaturate, throwing an apr_socket_recv connection reset by peer (104), meaning that the threads from the servers are heavy loaded with requests and can't handle any more requests.

Using the Apache AB testing tool, we have verified that the breaking point of the WAF_v2 has been estimated to be at the point where the service must handle 8091 concurrent requests. The WAF has experienced an 82,5% loss of capability to handle requests over the test progression, from 838 requests/second at the first reading to 146 requests/second, at the last reading. It is important to note that the bottleneck of the configuration is not located at the WAF, but at the backend servers. A solution would be to either change the loadbalancer distribution algorithm, or to vertically scale the backendserver nodes, to consider a performance improvement.

Security testing is intended to verify the overall effectiveness of the WAF security against undesired access of unauthorised users. Since system vulnerabilities can be originated at the application level of the service, or at the running environment of the service, like middle-ware or hardware components of the system, security measures must be taken in consideration when testing the WAF. Since the loadbalancers in our WAF, are configured with HAProxy software, we can take advantage of the features that HAProxy offers while configurating the haproxy.cfg file.

To verify the security, we have arranged and tested a loadbalancer's configuration to limit the rate of the web application to 10 requests every 10 second and we have arranged a configuration where there is a table to put in all the abusive IP's. Every IP that is marked as abusive is a threat or a false possitive. What we have observed from this test is that after the 10th request, in the 10 second interval, all the endpoint's TCP connections that were done after the 10th request and in the 10 second interval were rejected, as well as the inclusion of the IP to a blacklist of IP's marked as abusive. With this method, we are preventing denial of service attacks, rejecting new connections which can take down the loadbalancers and we are notifying the possible threats in real time to the engineers in charge of the service monitorisation. Every loadbalancer in the architecture has a statistics table, where there is a summary of the loadbalancer activity. This statistics web page is protected by a user and password authentification, since the site contains critical information about the configuration and activity of the loadbalancer, which only should be viewed by the system administrators.

9 Conclusion

The increase of global connectivity throughout the Internet to satisfy the quality of service from different companies' services, is becoming a necessity as the companies grow and offer more web application services. In order to satisfy the needs of a business growth, scalability is a must to meet market demands. In most cases, system scalability issues appear as performance problems caused by capacity limitation of servers and networks.
By building a high availability web application front end architecture, the architecture can be easily be vertical and horizontal scaled by adding more nodes and resources to the design, hiding communication, distribution and replication latencies as well as providing reliability and performance. When designing the architecture, we have not covered active-active replication in the backend servers, which is a must for data consistency in the backend side of the architecture, and we haven't covered network reliability and homogeneity which is necessary to keep the scalability consistent. What we did cover is network security and the WAF's development, from its analysis to its configuration, in order to satisfy the need of high availability.

With this in mind, scalability is one of the most important goals to achieve when building a high availability WAF, since a lot of requests are going to hit the architecture.

With WAF_v1, we have understood how we can deploy a web service on Docker, and how we can take advantage of an intermediary node to mask the internal implementation of the web service's backend, increasing security. The downside of using WAF_v1 for a high availability front-end is that in certain demands, the architecture will rapidly become a bottle neck to the web service and will probably fail.

WAF_v2 is an initial configuration of a high availability WAF, composed by two load balancers and three backend servers, as the use case for the analysis of this work. When referring to a initial configuration, we refer that every company service is different, with its different needs and goals. Depending on the use case, there must be a previous evaluation of what configuration would be more suitable taking in consideration the service's needs. After evaluating the use case, the idea is to configure WAF_v2 architecture to satisfy the needs of the use case service, by adding more nodes to the front end of the architecture, or by adding more resources to the load balancers, always to increase performance and high availability.

After analysing the results from the tests conducted to WAF_v2, it is important that the backendservers are given enough resources so there is a balance of performace between the WAF and the backendservers, preventing that the backendservers become a bottleneck to the service, as we have seen in the test results. An advantage of using the WAF_v2 architecture is that depending on the use case in which the architecture is deployed, the load balancers have been configured to prevent denial of service attacks by limiting the rate of HTTP requests accessing the architecture.

The loadbalancers have been configured for bot detection and IP masking and the load balancers could be configured to support SSL termination. WAF_v2 has passed all functional and non-functional tests, based in the initial requirements, satisfying the requirements from this work. The configuration for each loadbalancer from the WAF can be found in the haproxy.cfg file in the GitHub repository, which is the master configuration file for the configuration of the load balancers.

FUTURE WORK

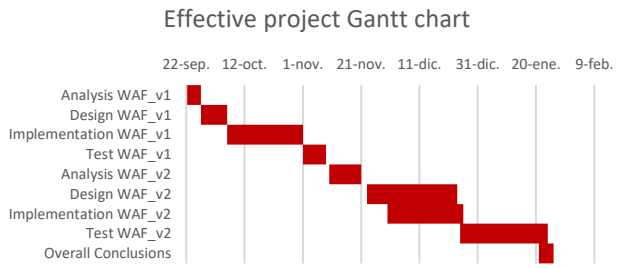

Fig. 10. Shows the Gantt chart from the planning followed to develop this work.

During the project development, we encountered with some issues that delayed the execution of the work. As we can see in Fig. 10, the development of WAF_V2 has lasted more time than expected. If we compare the effective gantt chart, which represents the initial planning of the different activities against time, with the ideal gantt chart, which represents the actual planning of the project development, which can be seen in Fig. 1, the development of WAF_v2 has experienced a 7-week delay. The causes of the delay were the research of knowledge to understand the architecure requirements and some changes in the initial design of WAF_v2 that had to be done in order to suit the requirements. Considering the comparison between the ideal project development and the effective and the current development of the project, the tasks that have not been accomplished will be done in future work, including: The development of WAF_v3, consisting of the same architecture as WAF_v2, but with the integration of the Elastic stack to the project and the development of a Kibana module to monitor logs and statistics in real time. The analysis of the comparison of performance gain/loss between the usage of different forwarding algorithms available by the HAProxy configuraton. Another line of work to be considered in future research, is the development of a WAF hosted in the cloud and the build of a simulation where the cloud services are provided to an organisation. Following the development of a WAF-as-a-service, is the comparison between WAF_v2 and the WAF hosted in the cloud to test which architecture would perform better in a certain use case.

ACKNOWLEDGMENT

This work was supported partly by Sigma Gestion Universitaria with NIF V61213641. Also, I would like to thank Sergi Robles Martínez, and Porfidio Hernández Budé, researchers from Universitat Autònoma de Barcelona, who provided insight and expertise that greatly assisted the research, although they may not agree with all the interpretations/conclusions of this paper.

REFERENCES

[1] C. Victor and H. Shahriar, "Web Application Firewall: Network Security models and Configuration" Tokyo, Japan. (Conference proceedings). 2018.

[2] Kemp Technologies, "High availability and Performance for Microsoft SharePoint" https://kemptechnologies.com/es/white-papers/high-availability-microsoft-sharepoint/. (Technical Report). 2019.

[3] T. Nick, Dell EMC, "High Available Data Protection with Dell EMC Islion Scale-out NAS" https://www.dellemc.com/ro-ro/collaterals/unauth/white-papers/products/storage/h10588-isilon-data-availability-protection-wp.pdf. (Tecnichal Report). 2019.

[4] Oracle, "Oracle Cloud Web Application Firewall"https://www.oracle.com/es/cloud/security/cloud-services/web-application-firewall.html. (Tecnichal Report). 2019.

[5] CloudFare, "La plataforma de nube global integrada" https://www.cloudflare.com/es-es/. (Tecnichal Report). 2020.

[6] Akamai, "Firewall de Aplicaciones Web" https://www.akamai.com/es/es/resources/web-application-firewall.jsp. (Tecnichal Report). 2020.

[7] Barracuda, "Web Application Security, Simplified "https://www.barracuda.com/waf-as-a-service. (Tecnichal Report). 2020.

[8] Amazon, "AWS WAF – Web Application Firewall" https://aws.amazon.com/es/waf/. (Tecnichal Report). 2020.

[9] FortiNet, "FortiWeb cloud WAF-as-a-Service". https://www.fortiweb-cloud.com/index/login. (Tecnichal Report). 2020.

[10] Barracuda, "Barracuda WAF-as-a-Service" https://assets.barracuda.com/assets/docs/dms/Barracuda_WAF_as_a_Service_SB_US.pdf . (Tecnichal Report). 2020.

[11] Guru-99, " Prototyping Model in Software Engineering: Methodology, Process, Approach, "https://www.guru99.com/software-engineering-prototyping-model.html. (Blog Report). 2019.

[12] R. E. H, "GitHub Repository" https://github.com/HernanEspinosa/Web_Application_Firewall.git. (Version Control). 2019.

[13] Hat. R, "Que es Docker?" https://www.redhat.com/es/topics/containers/what-is-docker. 2019 (Tecnichal Report) 2019.

[14] Cisco, "What is a VRRP?" https://www.cisco.com/c/es_mx/support/docs/security/vpn-3000-series-concentrators/7210-vrrp.pdf. 2019. (Technical Report). 2019.

[15] Docker, "What is a Container?" https://www.docker.com/resources/what-container. 2019. (Tecnichal Report). 2019.

[16] Elastic Products, "Productos de Elastic: Búsqueda, analíticas, logging y seguridad | Elastic " https://www.elastic.co/es/products/. (Tecnichal Report). 2019.

[17] https://httpd.apache.org/docs/2.4/programs/ab.html

[18] Docker Documentation, "Manage Data in Docker" https://docs.docker.com/storage/. (Technical Report). 2019.