BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

SECURITY SYSTEM FOR WEB APPLICATION ATTACKS ELIMINATION

BEZPEČNOSTNÍ SYSTÉM PRO ELIMINACI ÚTOKŮ NA WEBOVÉ APLIKACE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DOMINIK VAŠEK**
AUTOR PRÁCE

SUPERVISOR **Ing. KAMIL JEŘÁBEK**
VEDOUCÍ PRÁCE

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Vašek Dominik, Bc.**
Program: Informační technologie
Obor: Počítačové sítě
Název: **Bezpečnostní systém pro eliminaci útoků na webové aplikace**
Security System for Web Application Attacks Elimination
Kategorie: Bezpečnost

Zadání:

1. Nastudujte různé přístupy detekce nežádoucích požadavků na webové aplikace generované botnety.
2. S ohledem na nastudované přístupy navrhněte systém, který bude schopný detekovat a filtrovat nežádoucí požadavky z botnetů na webové aplikace. Systém se bude skládat z modulu pro reverzní proxy a detekčního serveru.
3. Navržený systém implementujte.
4. Na základě konzultace s vedoucím vyberte a implementujte filtraci alespoň jednoho typu nežádoucích požadavků.
5. Ověřte schopnost filtrace implementovaného řešení.

Literatura:

- SINGH, Karanpreet; SINGH, Paramvir; KUMAR, Krishan. User behavior analytics-based classification of application layer HTTP-GET flood attacks. *Journal of Network and Computer Applications*, 2018, 112: s. 97-114.
- ROVETTA, Stefano; SUCHACKA, Grażyna; MASULLI, Francesco. Bot recognition in a web store: an approach based on unsupervised learning. *Journal of Network and Computer Applications*, 2020, 157: 102577.
- KOCH, William; BESTAVROS, Azer. *Hyp3rArmor: reducing web application exposure to automated attacks*. Computer Science Department, Boston University, 2016.
- ALOMARI, Esraa, et al. A survey of botnet-based ddos flooding attacks of application layer: Detection and mitigation approaches. In: *Handbook of research on modern cryptographic solutions for computer and cyber security*. IGI Global, 2016. s. 52-79.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2 včetně.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Jeřábek Kamil, Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 30. července 2021
Datum schválení: 27. října 2020

Abstract

Nowadays, botnet attacks that aim to overwhelm the network layer by malformed packets and other means are usually mitigated by hardware intrusion detection systems. Application layer botnet attacks, on the other hand, are still a problem. In case of web applications, these attacks contain legitimate traffic that needs to be processed. If enough bots partake in this attack, it can lead to inaccessibility of services provided and other problems, which in turn can lead to financial loss. In this thesis, we propose a detection and mitigation system that can detect botnet attacks in realtime using statistical approach. This system is divided into several modules that together cooperate on the detection and mitigation. These parts can be further expanded. During the testing phase, the system was able to capture approximately 60% of botnet attacks that often focused on spam, login attacks and also DDoS. The number of false positive addresses is below 5%.

Abstrakt

Botnet útoky, které cílí na síťovou vrstvu, například poškozenými pakety a jinými metodami, jsou již v dnešní době úspěšně blokovány hardwarovými detekčními systémy. Pro aplikační vrstvu to však neplatí. V kontextu webových aplikací například vidíme, že útoky od botnetů obsahují často skutečné žádosti, které musí daný webserver zpracovat. Pokud se takového útoku zúčastní dostatek botů, může to vést k nedostupnosti poskytovaných služeb, popřípadě špatné funkcionalitě. To v konečném důsledku může vést až k finančním ztrátám, pokud je napadená stránka komerční. Tato práce navrhuje detekční systém, který je schopen detekovat tyto útoky z botnetů v reálném čase na základě statistického zpracování provozu. Systém je rozdělen do několika částí, které společně tvoří celou funkcionalitu a mohou být libovolně dále rozšířeny. Systém byl při testování schopen zachytit přibližně 60 % vážnějších útoků, které cílily často na spam a útoky na přihlašovací formuláře, ale také DDoS útoky. Počet falešně pozitivních adres byl při testování do 5 %.

Keywords

webserver, bot detection, botnet, ddos, mitigation

Klíčová slova

webový server, detekce botů, botnet, ddos, zmírnění útoku

Reference

VAŠEK, Dominik. *Security System for Web Application Attacks Elimination.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Kamil Jeřábek

Rozšířený abstrakt

Botnet útoky na síťovou vrstvu jsou již poměrně známé. V posledních dvaceti letech se již několikrát stalo, že velké nadnárodní společnosti byly na několik hodin nebo i dní bez připojení. Tyto útoky cílily na celou infrastrukturu, která vede k danému zákazníkovi. To však v dnešní době již není příliš možné, jelikož se používají hardwarové akcelerátory. Tyto akcelerátory slouží jako sondy na síti a dokáží takovéto útoky odfiltrovat mnohdy ještě dřív, než celý útok nabere takových rozměrů, že přehltí jednotlivé prvky v síti. [32]

Dalším problémem jsou pak botnet útoky na aplikační vrstvu. Tyto útoky byly pozorovány již na konci 90. let minulého století. Jedná se o legitimní dotazy, které ale nepochází od skutečných uživatelů. Cílem je, aby koncový server musel zpracovat co nejvíc dotazů a tyto dotazy zároveň nebyly odchyceny hardwarovými řešeními na síťové vrstvě. Nejčastěji jsou takto napadány právě webové aplikace, jelikož je poměrně snadné změnit cíl útoku. Mezi aplikační útoky řadíme DDoS (distribuované odepření služby), spam, útoky na přihlašovací formuláře, cross-site scripting, ale také stahování dat za účelem získání konkurenční výhody. Cílem těchto útoků je obvykle poškodit nějakým způsobem obsah daných stránek, nebo nepřímo způsobit finanční ztrátu. [32]

Cílem této diplomové práce bylo vytvořit systém, který je schopen v reálném čase detekovat botnet útoky na webové aplikace a zakročit, bude-li to nezbytné. Tento systém sleduje logy z webserverů a agreguje jednotlivé dotazy do časových oken. Z těchto oken jsou poté extrahovány jednotlivé atributy provozu, jako je počet požadavků, opakování požadavků, odezva, počet spojení, velikost dotazů. V první části systému jsou tato data porovnávána s kontrolním oknem, které reprezentuje historii jednotlivých atributů. Pomocí rozložení pravděpodobnosti pak sledujeme, jak jednotlivá spojení vybočují z normálu. Cílem je odhalit nadlimitní chování, které není u uživatele běžné. Jednotlivé atributy jsou poté ještě hodnoceny po skupinách, aby se zredukoval počet falešně pozitivních adres. Tyto adresy jsou nakonec předány do blokovacího modulu. Druhá část systému pak sleduje zatížení webserveru a podle toho dynamicky upravuje, jaké adresy budou blokovány. Výsledná konfigurace blokovaných adres je založena na době výskytu a odhalené závažnosti v rámci vyhodnocení.

Při testování systému byl záchyt botnetů nad reálným provozem okolo 60 %. Toto číslo je závislé na délce sledování. Systém si postupně buduje seznam blokovaných adres, který může dynamicky využívat podle zatížení webserveru. Při testování se ukázalo, že velká část botů se objevuje opakovaně. Počet falešně pozitivních adres se pohyboval do 5 %. V budoucnu je možné tento systém dále rozšířit o kontrolu uživatele a rozšíření způsobů zmírnění útoku. Je také možno využít funkcionalitu tohoto systému jako zdroj dat pro strojové učení a tím dále zvýšit detekční schopnosti.

Kapitola 2 probírá problematiku webserverů a botnetů. Kapitola vysvětluje využití botnetů a jak se botnety běžně chovají. Rozlišujeme zde vzory chování. Dále se také zabýváme způsoby ověření, zda je uživatel skutečný. Dále řešíme existující návrhy detekčních systémů a jejich přístupy. Tento systém je založen na poznatcích z [26]. Kapitola 3 poté rozebírá detaily systému a jednotlivých částí. Systém se skládá ze dvou částí, a sice proxy, která odesílá logy a přijímá seznam blokovaných adres a detekčního serveru. Kapitola dále vysvětluje ohodnocení jednotlivých atributů a také částí systému a jejich chování. Kapitola 4 vysvětluje samotnou implementaci systému. Popisuje třídní rozdělení systému, paralelismus a také řešení na straně Nginx webserveru. Konečně kapitola 5 popisuje výsledky testování celého systému. Celý systém je schopen zpracovat přibližně 125 000 dotazů/minutu v reálném čase.

Security System for Web Application Attacks Elimination

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Kamil Jeřábek. The supplementary information was provided by Ing. Martin Žídek. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dominik Vašek
July 27, 2021

Contents

Chapter 1

Introduction

With the ever-increasing number of users on the internet and the number of internet-connected devices, it is now easier than ever to overload the network or server infrastructure. This is especially true for web applications, which are usually available to the general public. There were attempts to hack or disrupt services provided by web applications since the rise of the internet. This can result in temporary or permanent loss of data, unavailable services and modified or stolen data. [32]

In the past decade, a number of botnets aiming to overload web applications by depleting available resources have been detected. The only effective defense against these bots is active monitoring and mitigation when an attack occurs. This can be done either by distributing the load over multiple servers or by detecting and disrupting traffic coming from botnets. The current detection systems are usually proprietary and expensive, since it is still cheaper to mitigate such attacks than to invest in a larger infrastructure that can absorb large botnet attacks. Other methods of detection consist of active monitoring by administrators or by using IP databases. However, both of these approaches are also expensive and therefore companies tend to look for other solutions. In this thesis, the goal is to detect these flood attacks in order to mitigate botnet traffic, such as distributed denial of service (DDoS) without the need for a robust distributed server infrastructure. [26, 32]

Chapter 2 defines the problems of botnets and botnet detection. The first part of this chapter aims to introduce the current trend in web application and webserver technologies. The second part focuses on botnets. It explains the problems created by botnets as well as the observable behaviour. Last part of this chapter introduces the possible detection approaches in individual steps. The first one is the attack detection. It aims to find suspicious traffic that resembles traffic from bots. This step is essential since additional services such as abuse databases or even log inspection from administrators are expensive. The second step is the challenging. It aims to sort out the bots by its capabilities, such as JavaScript support and interaction with the webpage.

Chapter 3 introduces a possible architecture of botnet detection system built upon approaches described in Chapter 2. The proposal consists of a system that implements the use of a Reverse proxy as a gateway and a Decision-making server that evaluates the traffic. The selected webserver implementation is Nginx. The proposal of the DMS is divided into modules for easier description. The first module is the Data module that aims to parse the requests and aggregate data for the remaining modules. The second module is the Performance module. This module aims to evaluate the change in webserver load in order to spot possible flood attack and increase the response of the system. The third module is the Attack detection. It evaluates the individual features of the traffic and sets

an evaluation for each user. Finally, the last module is the mitigation module. This module gathers the results from the previous modules and decides what action will be necessary.

Chapter 4 describes the implemented system. The first section of the chapter describes the communication between parts of the system. The second section describes the implementation of the Decision-making server (DMS) and its features. Lastly, the third section talks about the Nginx proxy and necessary steps for the incorporation to the system.

The testing of the system is described in Chapter 5. First, the log analysis is described. Then the observed detection capabilities of the system are demonstrated. Here, we look at the observed behaviour and results from the individual observed features as well as the optimal use-case of the system. Finally, the limits of the system are set based on the testing phase.

Chapter 2

Web application botnet detection

In order to better understand the issues of web application attacks, this chapter introduces existing webserver implementations, types of attacks and finally existing mitigation techniques.

2.1 Webserver

A webserver is a type of server that is used to interpret World Wide Web (WWW) requests. When talking about a webserver, it is important to respect the difference between the hardware and the software side of the server. In this thesis, one of our goals is to monitor the available resources, server load and the accessibility of services provided by the software side in order to evaluate the possibility of a DDoS attack.

On the client-side we can expect a web browser, which sends requests to webservers and interprets results from the server to users. Web browsers can be uniquely fingerprinted, which can be used for targeted detection of malicious users. There are also headless web browsers that lack graphical user interface and are used primarily for testing purposes.

The load on a webserver consists primarily of Hypertext Transfer Protocol (HTTP) and Hypertext Transfer Protocol Secure (HTTPS) requests. Upon receiving these requests, the server-side then initializes a series of tasks in order to complete the request resulting in asymmetric work load between client-side and server-side of the web. This can be used to overwhelm the server infrastructure by a small number of clients.

Based on the requests, the server-side can perform time-consuming operations, which can result in slow response time or even inaccessibility of provided content. Such requests consist of:

- request Structured Query Language (SQL) tasks;
- disk writes/reads;
- backups.

In modern webserver implementations, we can encounter load balancing modules that can decrease the server load by dispersing the traffic across multiple webserver instances. Other than that, we can also encounter modules which allow traffic policing in order to maintain availability and accessibility of services for legitimate users. [17]

In Figure 2.1, we can see the most common webserver implementations and their market share over time. As we can see, the Nginx webserver is steadily on the rise, slowly taking over the Apache webserver.

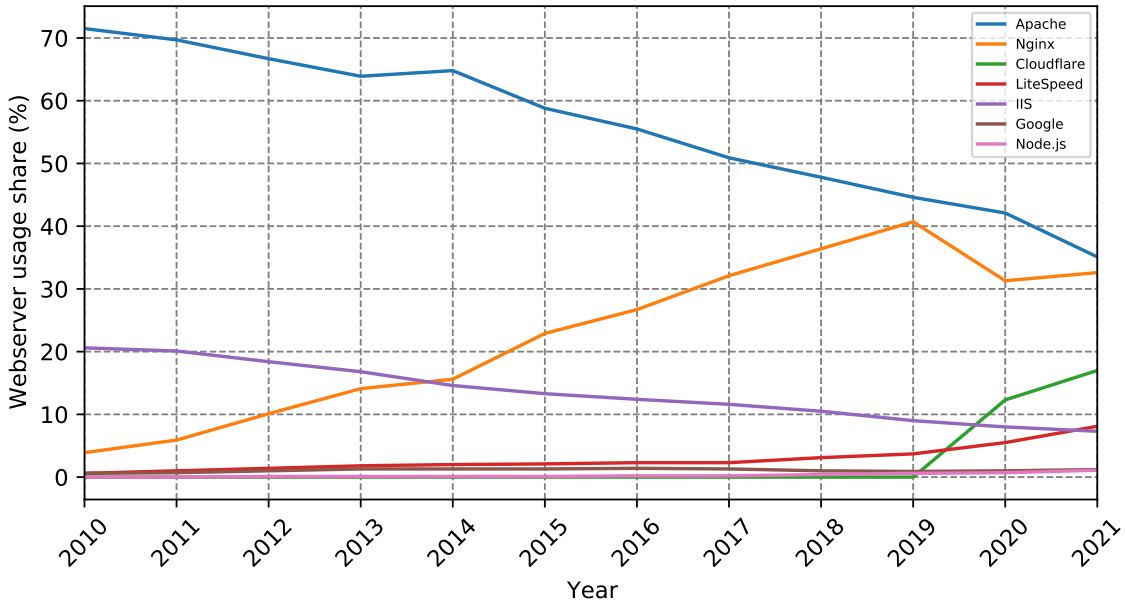

Figure 2.1: Share of webserver implementations on the internet. [30]

As was mentioned earlier, the webservers have asymmetric distribution of workload between client and server. Even if this was not the case, considerably more clients requesting access to the webserver than webserver instances running can be still expected.

Under normal conditions, the administrator can scale the server infrastructure for the workload created by usual user's traffic. This can be done by observing the workload over a long period of time and allocating appropriate resources. However, there are some instances where the amount of traffic can exceed designed traffic/server capabilities, thus resulting in downtime or inaccessibility of services. Such instances can consist of a flash crowd or a DDoS attack created by a botnet.

A flash crowd is a term describing a sudden influx of traffic on a webserver that is not common, but consists of legitimate users. In contrast to illegitimate traffic from a botnet, it is desirable to process this traffic even though it might increase the response time for all users.

DDoS are attacks that consist of traffic from illegitimate users with the intention to disrupt services provided by the webserver. [13]

2.1.1 Web applications

Web application is an application that runs on the webserver. These applications usually provide a service to the end user. The two main trends of web applications are classic websites and web APIs. A website is a program that utilizes the webserver as well as the browser on the client side, providing service for the users through GUI. A Web API provides an interface which can be used for communication between multiple applications.

Typical use-case of a web applications is:

1. User sends a request to the webserver.
2. Proxy sends the request to the corresponding webserver.
3. Webserver performs the tasks contained in the request.
4. User receives the results.

Web applications have many advantages. The main one being that web applications are platform independent. This means the software engineer implementing the application does not need to understand the hardware and software of the client. Additionally, there is no need to handle different versions between clients, which can save money by limiting the amount of maintenance and support required.

Web applications are a common business model today, since they are cheap to develop and maintain. Furthermore, they are easy to distribute towards the users and are fast and reliable. [7, 28]

2.1.2 Webserver implementations

Currently, there are over a dozen of webserver implementations. However, over 90 percent of websites use one of the following implementations [30]:

- **Apache**

  Apache is an open-source webserver developed by Apache Software Foundation. There are many existing open-source modules available. Some of its core functions are high scalability and event-based evaluation. It is currently the leading implementation amongst webservers.

- **Nginx**

  Nginx is the second most popular implementation of webserver. There are two versions, namely Nginx open-source and Nginx Plus, which is developed by Nginx, Inc. Nginx was made with the intention to replace the older Apache webserver that was known for high memory consumption and low request throughput.

- **Cloudflare**

  The Cloudflare webserver is a proprietary implementation owned and maintained by Cloudflare, Inc. It is deployed on Cloudflare Content Delivery Network (CDN) and currently hosts over 17 percent of all websites.

- **IIS**

  IIS is solely Windows based proprietary webserver implementation from Microsoft. The IIS has been overtaken by Nginx, dropping from second most used webserver to fourth place in the past 10 years.

2.2 Botnet

Botnets are a number of internet-connected devices that can perform a common task such as a DDoS attack. Most botnets consist of devices like routers, IoT devices, computers

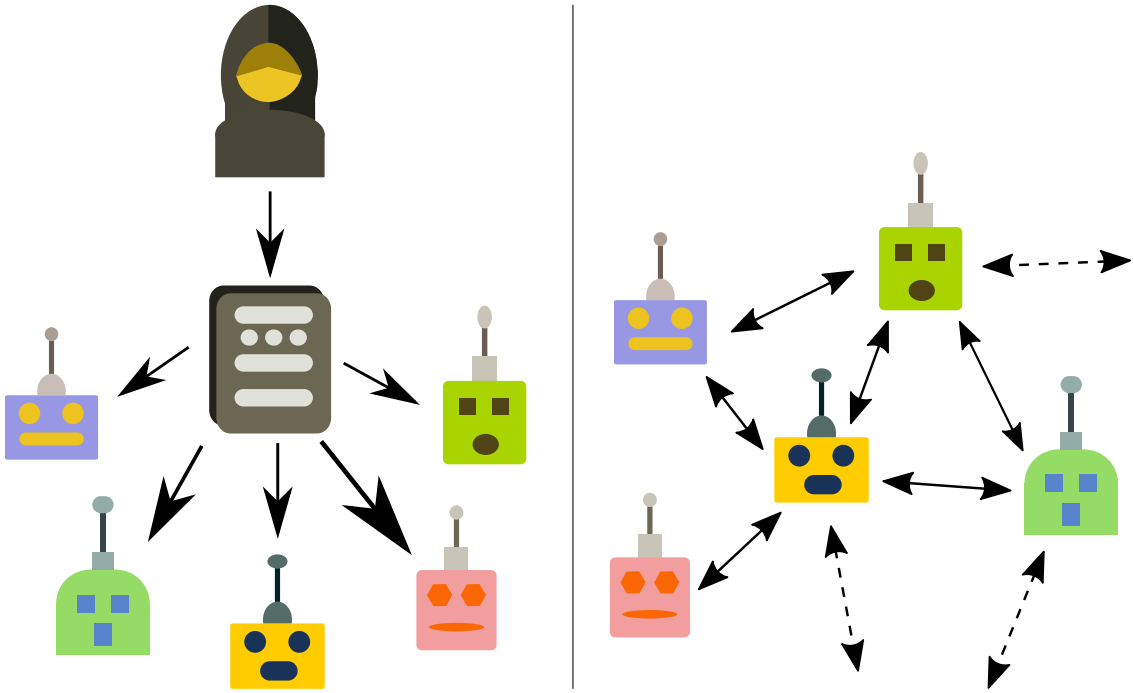

Figure 2.2: Botnet infrastructures: on the left side is the client-server architecture; the right side depicts the peer to peer architecture.

or smartphones infected by malware that is controllable by a perpetrator. With the ever-increasing size of botnets and its potential to disrupt services or cause financial loss to businesses, it is important to distinguish traffic generated from these botnets from legitimate user traffic.

There are two types of botnet infrastructure as depicted in Figure 2.2. On the left side we can see the client-server architecture and on the right side the peer-to-peer architecture.

Client-server model uses command and control servers that distribute requests to all bots and are controllable by the botmaster. The other approach is peer-to-peer botnets, which are more resilient to disruption as bots can communicate directly with each other.

A bot is a program that can perform some kind of automated work. We can distinguish between good bots and bad bots. In web applications, an example of a good bot is a web crawler, which makes it easier to find information on websites. On the other side are bad bots, which aim to disrupt services provided by a website. An attacker can use a botnet to deploy an attack by a large number of different bots that can also focus on different applications running on victim's side. In case of web-server applications, there are currently four distinguishable generations of bots. With each generation comes increased complexity of available tasks and detection methods. [23]

2.2.1 Botnet applications

A botnet can have a variety of use-cases. An example of a good use-case is web scraping or indexing of the world wide web. In contrast are use-cases where botnets are used for malicious actions such as spam, data theft, validation of leaked data or distributed denial of service. Some of the popular botnet use-cases are further described below:

- **Web scraping**

  Web scraping is used to gather information contained on the website. Such information can consist of e-shop prices, available products or other information. Even though web scraping is not an attack, it is not always desirable for web administrators since competition can use this data and cause a loss of revenue. [6]

- **Web indexing**

  Web indexing is similar to web scraping, but aims to gain different information. Web scraping is focused on obtaining specific data for some specific use-case. Web indexing on the other hand focuses on indexing websites. This is usually done for easier searching of the website. Web indexing is done by search engines such as Google, Seznam, Bing, etc. [6]

- **Spam**

  Some bots are made specifically to spread disinformation. Botnets can be used to spread fake news across the web. This is usually hard to detect without challenging the user, since the botnet can send one message and then go offline. [11]

- **Data theft**

  With the increasing complexity of botnets, it is possible to use a large botnet to hack websites. The botnet can use known platform dependent security flaws in order to steal sensitive data while staying undetected. These flaws and vulnerabilities can be later used by the hacker. [1]

- **Cracking**

  While botnets can be used to steal data, they can also be used to decipher the credentials of users in an attempt to steal their account. This can be achieved either by using the stolen hashes of user passwords, which decreases the complexity of hacking an account, or by using the large-scale platform of a botnet for a brute force attack. [1]

- **Scripting**

  Botnets can also be used for finding vulnerabilities in websites that can be later exploited by hackers to implement malicious scripts on the attacked webpage. This can be done for example by checking for XSS injections. [1]

- **Distributed Denial of Service**

  DDoS is another type of botnet attack that aims to disrupt services provided by victim, which can result in loss of revenue or publicly available data. We can distinguish two methods of DDoS attacks.

  The first one is vulnerability attack in which the perpetrator sends malformed packets to the victim in an attempt to crash an application or used protocol. However, this type of attack is primarily avoidable by fixing vulnerabilities.

  The second method aims to starve the victim of resources. This is usually done by sending a large number of requests to a specified target on a scale that is not manageable by available resources. We can divide such attacks into two groups. The first one is network layer DDoS; the second is application layer DDoS. Figure 2.3 depicts a DDoS attack leading to inaccessibility of services for legitimate users. [25]

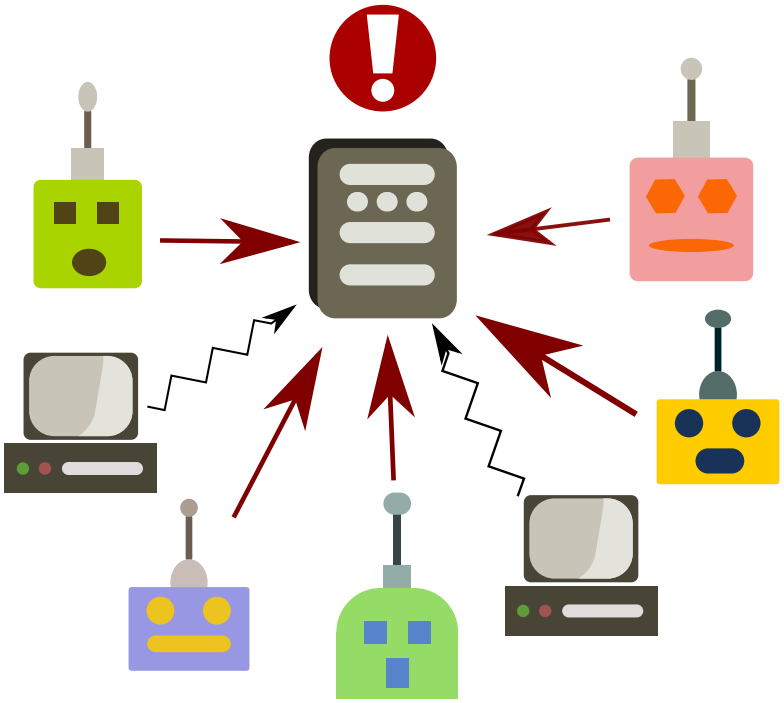

Figure 2.3: A DDoS attack on a server: Due to the illegitimate bot traffic, the webserver is overloaded, resulting in inaccessible services for legitimate users.

DDoS does not focus on application layer only. Other than the vulnerability and starvation of the application layer, the division is also between which TCP/IP layer is under attack. Namely, the division is between the network layer and the application layer. In the network layer, the perpetrator aims to overload the network infrastructure of targeted victim. This is usually done by packet flooding of targeted network in the magnitude of hundreds of gigabits per second. The most common types of attack include ICMP and IP protocols. Nowadays, these types of attacks are mostly mitigated by hardware infrastructure such as ASIC/FPGA firewalls. [21]

Denial of Service on Application layer consists of resource-consuming tasks requested by perpetrator. These requests consist of legitimate user traffic, thus can pass through network layer DDoS protection. The perpetrator aims to drain server resources as mentioned in Section 2.1. This can lead to slower response times for legitimate users or even unavailability of services [32].

Distributed DoS attacks focus on the need to process requests by the endpoint in larger quantities. Since these attacks are distributed over a large network of fake clients, they can even perform standard requests in order to masquerade as regular user traffic. Therefore, these distributed DoS, similarly to classic application layer DoS, consume available resources. This can cause slow response time or even inaccessibility for legitimate users, which can lead to loss of revenue, loss of data, etc. [21]

2.2.2 Botnet attack strategies

HTTP botnet attacks usually consist of some common pattern that is distinguishable from legitimate user traffic. This is further discussed in [26], where different botnet patterns

are described. This distinction is also proposed by Mazebolt [14]. The main reason for the comparison of flood strategies is that most flood attacks are under the threshold of rate limiting. Thus, the detection system has to look at the traffic patterns. In this section, we will discuss the patterns of botnet traffic. These patterns consist of GET Floods, which are spread across multiple addresses. This is possible because the attacking bots are usually part of a botnet and therefore can distribute the otherwise over-the-limit patterns across multiple addresses. Bots are not always capable of behaving the same way as legitimate users and miss the randomness of legitimate users.

The HTTP Flood can be distinguished by the request itself. The flood consists of requests using the HTTP/HTTPS protocol. Most common are GET and POST requests. HTTP floods usually lead to inaccessibility of the server and thus loss of revenue. This usually happens due to DDoS, however even scraping bots or spam/sniffing botnets can cause such outcome. [32]

Some of the more common patterns are described below:

- **Dynamic**

  Dynamic flood changes the requests for the webserver. This is done for example by adding randomly generated suffix to the URL or traversing the website (similar to scraping). Using this approach, the flood attack can hide some statistical features of the traffic. [14]

- **Constant**

  This type of flood attack has constant request rate and is simple to detect even at lower request rates. The attack is based on sending requests at a constant rate, which can consist of one request sent periodically or multiple requests. This is usually done in an attempt to stay under detection of DDoS prevention systems by keeping the request rates low. [26]

- **Random**

  The bots generate requests in random time intervals, thus making the detection harder. The goal of this strategy is to avoid detection by time intervals by sending random bursts of traffic throughout the attack duration. [26]

- **Slowloris**

  Slowloris, also known as low and slow is a botnet attack that aims to obtain all available resources of the webserver. The idea is to slowly open thousands of connections and keeping them alive by only a few requests per minute. If enough bots partake in this attack, it can be undetectable especially in the beginning. [32]

- **Flash**

  The idea is to momentarily overload the webserver. The goal is to make an event similar to flash crowd and try to skip past the detection mechanisms of the server. This pattern can also be used for establishing the system limits. [26]

- **Same page**

  The botnet continuously sends requests for one page. The botnets can try to hide from intrusion detection by staying in popular webpage set and therefore maintain the attack for longer periods of time. [26]

- **Continuous**

  Requests with high workload on the webserver are sent by the botnet in low rates, aiming to slow the response time of the webserver. [26]

2.2.3 Web-based bot generations

Radware [23] and Datadome [5] both describe the evolution of botnets using generations as described in this subsection. There are currently four distinguishable generations. Each new generation is harder to detect and is practically undetectable by methods for earlier generation bots. A detailed description of mitigation methods follows in Section 2.3.2.

- **First generation**

  First generation bots usually consist of a series of scripted tasks for cURL/Wget requests. Other than for the DDoS attack, they are also often used for web scraping.

  Mitigation: These bots are easy to identify as they do not use cookies or JavaScript. In order to mitigate such attacks, it is necessary to either block IP addresses or use user agents.

- **Second generation**

  Second generation bots are based on headless web browsers such as Chrome or Firefox. The main difference in contrast to first generation is that these can utilize cookies and also execute JavaScript tasks. This generation is often used for automatic testing but can be also used for skewing statistics.

  Mitigation: Unlike the first generation, this generation can perform JavaScript tasks; however, it can be identified through its browser and device information. In case of an attack, similar attributes across the botnet can be expected. Even though headless web browsers are usually detectable by some common characteristics, an attacker can attempt to hide these characteristics. Therefore, if the recognition fails, it is optimal to send captcha.

- **Third generation**

  In contrast to previous generations, it is impossible to distinguish these bots without observing client's interactions. This generation employs fully operational web browsers and can emulate human-like behaviour based on linear mouse movements, keystrokes, etc. However, they are missing the randomness of human interaction.

  Mitigation: It is necessary to implement a captcha challenge or behavioural analysis comparing to legitimate user traffic.

- **Fourth generation**

  The last generation is almost indistinguishable from legitimate user traffic. These bots show randomness similar to human behaviour. Additionally, a periodic change of user agents and source IP address is expectable from these bots, which makes mitigation harder. Fourth generation bots can use low and slow attacks in order to pass through mitigation methods.

  Mitigation: To combat these bots, it is necessary to implement intent-based behavioural analysis and detect differences between legitimate user traffic and bot traffic.

2.2.4 Known botnets

Botnets usually attack by attempting to use as many bots as possible to send requests to the attacked domain. The use of botnets can be bought in order to commence an attack on specific websites. The name of the botnet is often decided by the malware that created the bots, however different entities can control these bots. Some of the well-known botnets are:

- **Mirai**

  Mirai is a botnet that runs on Linux devices and primarily targets smart home devices. The malware is spread by scanning IP addresses and attempting to log into poorly secured devices that are accessible from outside using default login credentials. The number of infected devices with the Mirai malware is in the millions. [4]

- **Chameleon**

  Chameleon is a botnet that runs on Windows devices. At least 120 thousand devices were detected as infected. The goal of this botnet was to generate traffic similar to user traffic, generating ad revenue. The botnet also included primitive user interaction. [24]

- **Cyclone**

  Cyclone botnet is known for killing other bots from the system it infects. Mostly used for HTTP Floods. [8]

- **Nitol**

  Botnet that runs on Windows and is located mostly in China. In 2012, many computers sold in China were infected by this botnet. In total, over 50% of all botnet traffic in 2012 came from this botnet. Nowadays, there are several other botnets based on this one. [8]

2.3 Filtration methods

Botnet detection is a complex problem. Hard baselining, which consists of setting hard limits for specific features of the traffic, is not the solution and should be used only when the attack is already occurring, since this can lead to a large number of false positives. In order to correctly classify the traffic, a soft decision should be made first. Such soft decisions can be made by observing a series of parameters of the user traffic, which is then evaluated and further action, such as challenging, is carried out where necessary. This section discusses possible ways of bot and botnet detection.

2.3.1 Attack detection

Attack detection is the first step in finding botnet attacks. The goal of attack detection is to flag malicious traffic. While this results in false positives, these clients can be further challenged. This can be done through bot detection techniques, which is further discussed in Subsection 2.3.2.

In modern systems, attack detection is done either through monitoring and active intervention from the administrators, or by specialized systems that attempt to detect and mitigate such attacks. It is, however, not always possible. Additionally, proprietary implementations can be black box, which means that the client does not always know if the reported

false positive rate is accurate or if the system is effective. Allowlisting and denylisting can be also used for decision-making in some cases. Many bots are defined as good bots, such as web crawlers, that are essential for websites in order to gain user awareness. Similarly, there are many bots operating on specific IP address ranges that can be mitigated by denylisting.

There are many approaches proposed and tested in research papers considering the attack detection and mitigation of botnet attacks. Usually, it is done either through machine learning or statistical evaluation and dynamic baselining. Even though it is possible to create a heuristic analysis, such analysis is not suitable for botnet attacks since they are always evolving [34]. The following section introduces some of the existing proposals.

Similarity-based approach

This method [31] takes into consideration flash crowds that can be hard to distinguish from botnets. It is done by considering the need for botnets to be either as large as flash crowds, or more commonly, use higher request frequency or request large payload. Another important attribute is similarity between flows, where botnets often consist of a single preprogrammed sequence of tasks to complete. Based on the similarity, the authors created the flow correlation coefficient, which can distinguish botnets from flash crowds with high accuracy and was based on four features: popularity, request rate, transition probability from one page to another and object size. However, even though it had promising results with high accuracy, the solution was not suitable for realtime use and was unsuccessful in distinguishing flash crowds from botnets.

HTTP GET requests per IP address (HRPI)

Another method, HRPI [21], looks at the chain of successive HTTP GET requests. It takes into consideration the number of HTTP GET requests over a time period per IP address. Usually, the distribution of IP addresses is scattered over the internet, in case of botnets, it was observed that these infected addresses can be clustered. This method can be used to distinguish a flash crowd from botnet attack. Kalman filter is used to process the traffic matrix as a whole and make the estimation. The previous time interval is then used in contrast to the new interval. Lastly, these data are used for classification in a Support Vector Machine trained by Adaptive Auto-Regressive function. This method achieved high detection efficiency and flexibility.

Detection using Analytical hierarchical process

In contrast to most methods which mainly focus on traffic patterns extracted directly from request headers, [27] proposed to look into log files of webservers. This method gathers relevant information from logs that are then used for deciding whether or not the attack occurred. Detection of suspicious user activity is done through Dempster-Shafer theory of evidence. This method also brings low processing time and high accuracy with low false alarms. The results are similar to other existing approaches.

SENTRY

SENTRY model [33] aims to reduce the application-layer DDoS through challenge-response approach. The approach attempts to analyse the physical bandwidth of an attacker, dynamically adapt to various workload scenarios and finally block suspicious requests. The client

requests are first sent to a moderator which challenges the client. The client then answers with its bandwidth resources, which are then compared to the actual request. If the challenge is successful, the client's further communication continues directly with the webserver. This method can therefore effectively capture slow read/send attacks.

User behaviour analytics-based classification

Last but not least, [26] proposed to observe four characteristics that would differentiate bots from legitimate users. The detection is based on log evaluation, similarly to [27]. The detection is based on two sets of time windows, large time window of 120 seconds and small time window of 30 seconds. The large time window consists of four small windows; therefore, the attack detection is 120 seconds. The detection focuses on machine-learning strategies, where it observes several bot-specific behaviours. The authors propose that in order to exhaust webserver services, botnets usually need to increase their request rates in contrast to legitimate users. Another proposed criterion is response index, which considers the difficulty to process a user request. Popularity index is used to observe popularity of web pages, since 10% of all web pages receive 90% of total traffic [2]. Legitimate users usually request these webpages. Botnets on the other hand do not know this subset of hot pages, which can lead to different browsing distribution in contrast to legitimate users. Finally, the last observed criterion is repetition index. Legitimate users are not keen on requesting same pages multiple times in a short period of time. These observed strategies alongside machine-learning provided a detection rate of up to 99%.

2.3.2 Bot detection

In modern web applications, we can always expect some sort of a bot to browse our website. Since not all bots are malicious, we should first distinguish between good bots and bad bots. Web crawlers such as Googlebot used for website indexing usually publish their IP address range. These bots are hosted with their own IP address. Thus, we can use an IP address of these bots as a deciding factor by allowlisting these addresses.

Bad bots, on the other hand, can use IoT devices and other infrastructure, which can use NAT, VPN, or other proxy services. This means that in contrast to known bots used for indexing, it is not always feasible to use an IP address for distinguishing these bots. In this case, we can use browser fingerprinting. The fingerprinting returns a unique identification of the attacking device. However, some malicious bots are hard to fingerprint, which results in the need to denylist attacking IP addresses. Some companies also provide databases of infected IP addresses, which can be further used for denylisting. Anyway, these databases are usually expensive to use, and therefore it is essential to use such services only when necessary.

When an attack occurs, we should check user-agents for cookies, JavaScript support, web browser characteristics, and finally observe the behaviour of clients. This is done in order to determine if the client is a bot or a legitimate user. Both Radware [23] and Datadome [5] describe the mitigation methods for individual bot generations as below:

- **Cookies**

  Nowadays cookies are mostly used to quickly tell web browsers or user sessions apart. Cookies are stored in a web browser and can be requested by webserver at any time. Because of this, checking if a web browser supports cookies is the simplest and least resource consuming way to check for simple bots such as first-generation bots.

- **JavaScript**

  Modern web browsers rely heavily on JavaScript. JavaScript is being interpreted at client-side; therefore, it can be used for a large variety of tests. The initial test for JavaScript is to check whether or not the web browser even supports JavaScript, since first-generation bots do not support it.

  Captcha is another use for JavaScript. It works by challenging bots to do simple tasks that would not be a problem for a human. Captchas can also measure the time it took to be completed. This method is often used for challenging third-generation bots and is often accompanied by movement pattern tracking in order to better tell apart human and bot behaviour.

- **Web browser characteristics**

  This method is used for headless web browsers. Since second generation bots support JavaScript, we cannot rely on simply checking JavaScript functionality. Rather, we can check the characteristics of the web browser.

  The characteristics used to tell headless web browsers apart from standard web browsers are the following [10, 29]:

  - User-agent contains string representing headless browser;
  - Application version;
  - Navigator.webdriver unset;
  - Available plugins;
  - Inconsistency between notifaction.permission and navigator.permissions.query;
  - Time elapsed;
  - Outer dimensions of webbrowser set to zero.

- **Interaction based**

  A large portion of bots is able to interact with websites through user interface. The interaction however is rather simple. Most bots use straight lines and quick responses to send requests in contrast to legitimate users. With observation of the user interaction, it is possible to distinguish these bots from legitimate users without the need for deeper understanding behaviour of users.

- **Intent based**

  Modern bots often manifest human-like behaviour. Primarily, they add randomness of human interaction into their interaction with websites. This makes them harder to detect. Simple interaction-based analysis is often insufficient, since it focuses on the random interaction behaviour of the user. For successful detection, it is necessary to observe the user behaviour over a long period of time in order to determine the purpose of the client visiting the website. This is often done in collaboration with semi-supervised learning that aims to accordingly classify human interactions versus bot interactions. [22]

2.3.3 Mitigation

We can use several approaches for mitigation of botnet attacks. Some of the mitigation methods are dropping traffic or traffic policing. Traffic dropping consists of dropping all connections from specific user. On the other hand, traffic policing consists of slowing the connections or cutting over-the-limit traffic. This allows access to suspicious users while decreasing their impact on the webserver. Another mitigation method is to set a threshold per IP address. This might be problematic with mobile network traffic, since it heavily uses NAT. The goal of the mitigation is to reduce the number of requests on webserver. [15]

For a large number of webserver instances, we can distribute server load across multiple servers. This can also reduce the impact of a botnet attack. If combined with an attack detection system, we can combine distribution of traffic with traffic classification. This results in lower risk of false positives, since we can serve clients that behave like bots. Even though this can result in higher delay for false positives, we are usually still able to serve their requests while maintaining stability for legitimate users that are correctly classified by the detection system. [17, 16]

Chapter 3

Proposed detection system

The previous chapter focused on the problematics of webservers and botnets and the necessary steps for the accurate distinction of bots and humans. This chapter discusses proposal of a system based on the obtained information and also what will be necessary in individual parts of the system. In the first section, we discuss the architecture of the system. The second part discusses the system in further detail, proposing the implementation.

3.1 Architecture

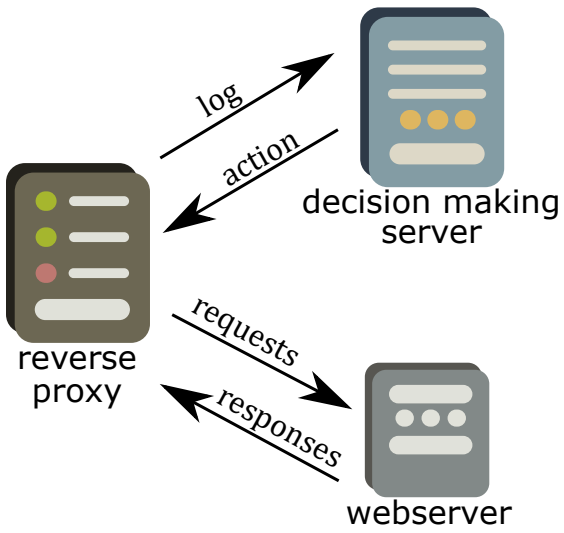

Figure 3.1: Architecture hiding the infrastructure from users behind the RP, consisting of a webserver and a DMS.

There are several approaches for the implementation. One possible approach consists solely of a webserver module that would implement the filtration and detection of all traffic. Although this approach is easy to implement, it is not quite scalable. Additionally, keeping the decision-making on the webserver increases the load on the server, which could increase overall latency. The other approach is to use a reverse proxy (RP) alongside a decision-making server (DMS). This solution is depicted in Figure 3.1. The RP consists of a webserver module, such as a Nginx module that would send necessary traffic infor-

mation to the DMS. Based on these data, the DMS can classify the traffic and return corresponding response for the traffic to the server running the RP.

3.1.1 Reverse proxy

Reverse proxy is a type of server that receives traffic from the clients and transfers it to corresponding webserver, as we can see in Figure 3.2. In contrast to forward proxy, which is in front of the client, the RP sits at the server-side and thus can be usually managed by the administrator. A good use of RP is when we want to hide the webserver infrastructure, since clients think that all traffic comes from the proxy. In fact, the traffic traverses through the proxy to the corresponding destination. [3]

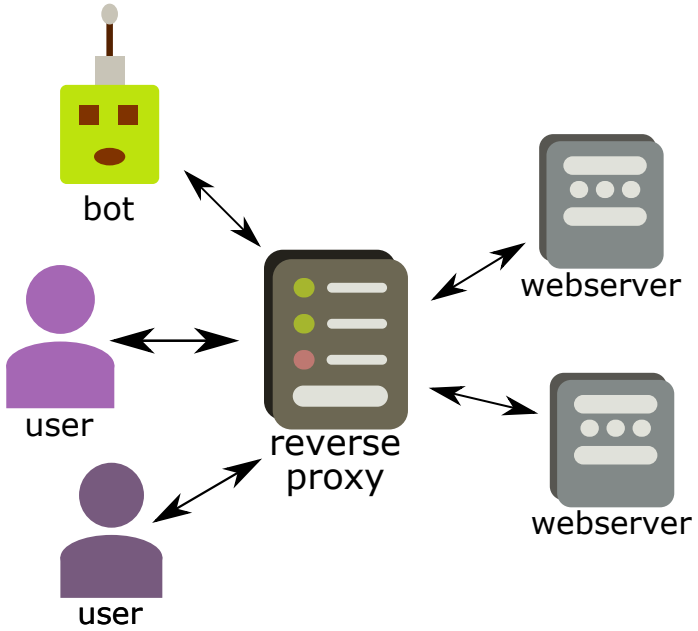

Figure 3.2: An example of RP: the traffic is being passed through the RP, hiding the infrastructure behind.

The RP can be used in various scenarios such as load-balancing, web acceleration through data compression and caching, as well as security, where it can be used as a firewall hiding infrastructure or filtering incoming traffic. This is especially valuable since it decreases the number of time-consuming requests sent towards the webserver. Because of this, a good approach for traffic classification would be to use a RP in front of the webserver, which can pass the traffic through or send additional information about the traffic towards a DMS server that would classify the traffic and decide the next steps. [20]

Nginx is a webserver engine that has over the last decade became one of the leading webserver implementations. It can be used in various scenarios, such as a RP, load balancer, mail proxy, or HTTP cache. Furthermore, the Nginx Proxy is also compatible with Apache.

Although not being the first webserver implementation, Nginx was created to address the c10k problem. It is a problem that comes from creating threads for individual clients and fast exhaustion of available hardware resources. This problem made it difficult for webservers to simultaneously serve more than ten thousand connections. While Nginx uses threads, it is built on event-driven architecture. This means that multiple clients

can be processed by same thread. The goal is to use non-blocking operations with common threads where possible and create new threads for blocking operations [12]. Other than asynchronous processing, Nginx also brings large scalability thanks to load balancing methods where multiple Nginx instances can be running across multiple servers providing the same content. [19]

Nginx is available in open-source and plus versions. While both versions can use community implemented modules, the plus version already contains many additional functions. Both implementations consist of load balancer, webserver and RP.

Currently, the Nginx RP is available for plus and open-source versions. Since it is already implemented, a good approach would be to use the existing implementation and further expand it by a module that would add necessary functions for the DMS [18].

Since this thesis is about keeping the hosted websites available for clients, it is also important to mention the Nginx load balancing methods in more detail. The load balancing is used to distribute load from clients across available webserver instances with the goal of reducing latency and serving all clients. Server weight can be set for some balancing methods, taking server resources into consideration. Without additional modules, there is built-in support for several balancing methods [17]:

- **Round robin** – This is the default method which distributes the load evenly.
- **Least connections** – Sends new requests to the server with least active connections.
- **IP Hash** – Resulting webserver is decided based on hash of clients IP address.
- **Generic Hash** – Like IP hash, but the hash string is custom and can take multiple client's attributes into consideration.
- **Least Time** – This method is available in Plus version. Webserver is chosen based on its latency.
- **Random** – The webserver is chosen by pseudorandom function based on administrator settings.

3.1.2 Decision-making server

The goal of the DMS should be to gather traffic information from proxies or individual webservers and use it for decision-making. The chosen action can be either one of the mitigation techniques, where the DMS sends request to the RP to halt or slow incoming traffic for some malicious users, or to further challenge the user. The challenge part could be done by an external service, such as captcha, since thanks to the DMS, most of the traffic will be cleared from the challenge part, reducing the costs of challenging using external services. Other method could be to create a custom challenge module that would challenge the clients where necessary with help of the webserver.

3.1.3 Infrastructure

The proposed architecture as seen on Figure 3.3 consists of a RP, a webserver and a DMS. The goal is to observe and perform the mitigation outside of the webserver. This way we can reduce the load of the webserver without the need to interfere with existing configuration.

In the first step, we want to add a RP that would send all traffic upstream to the webserver, if it is not already present in front of the webserver, but also send access logs

to the DMS. The DMS can later also use the proxy for its mitigation techniques, since the user sees only the proxy.

The DMS consists of four main modules. The data module should export data from the syslog messages, attack detection module should classify the traffic, performance monitor module should observe the webserver load and finally the mitigation module, whose main purpose is the decision-making based on the available information. The mitigation module can also invoke the Challenge module, which should challenge the flagged users.

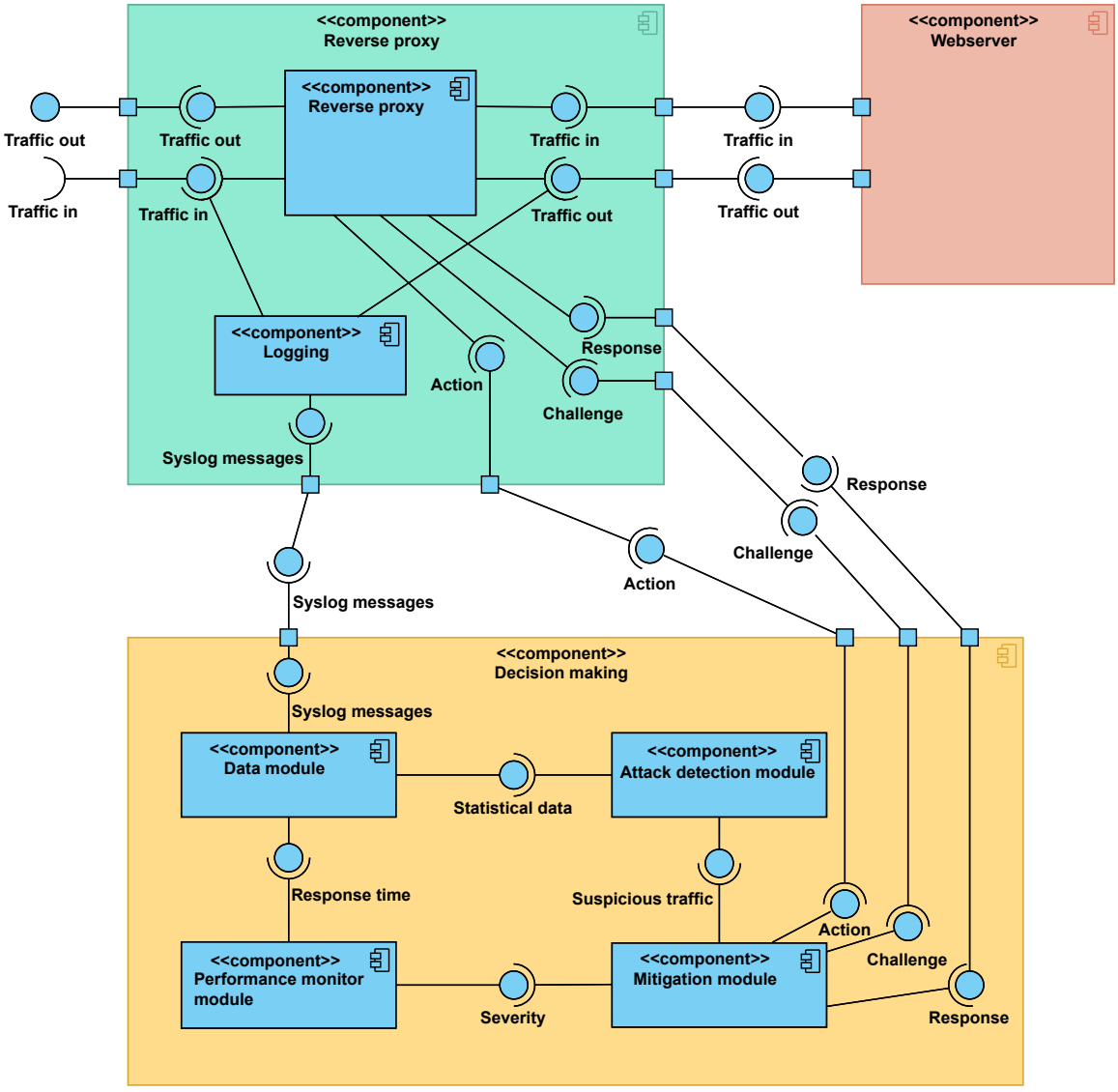

Figure 3.3: Diagram of the proposed infrastructure.

3.2 Proposed implementation

In this section the parts of the system are further decomposed depicting steps that are necessary for the implementation. Furthermore, some additional features that could increase the overall efficiency of the system are also discussed.

3.2.1 Reverse proxy

The proxy module is already implemented in the Nginx environment, as well as the syslog module. However, as we can see in Figure 3.3, the DMS is expected to send requests for the RP. The requests consist of user challenges that can be done through the RP, but also blocking of flagged traffic. In this case, we can use the implemented `http_access_module` and rate-limit module of Nginx. Nevertheless it will be necessary to expand these modules with an interface that can be controlled by the DMS. Such expansion can be also done by an external script that would feed the necessary data to these modules. This approach will increase the sustainability of the code, since maintained modules will be used.

3.2.2 Decision-making server

The main part of this proposal is the implementation of the DMS. The proposal can be seen in Figure 3.3, more detailed version is described in the following section. In this proposal, the system is divided into four main modules to easily demonstrate the individual functionality of the system.

Data module

As we can see in Figure 3.4, the DMS should consist of several modules that should cooperate on the botnet detection and mitigation tasks. The first, data module, consists of a syslog server that can receive syslog messages from multiple reverse proxies. In best case, we should be able to serve multiple reverse proxies with one instance of DMS. The accuracy of evaluation is based on the amount of legitimate user traffic contained in the logs. Therefore, we should first strip bots that report themselves as bots from the evaluation. Another part of the data module is the data parser that will extract all necessary traffic information from the logs for individual tests. These data will be then split into time windows that will be sent into attack detection module to be evaluated. The parser will also provide information about the performance of the webservers and the changes in active connections. Other data, such as IP addresses, will be sent to the mitigation module. Response time and connection information will also be sent to performance monitor module.

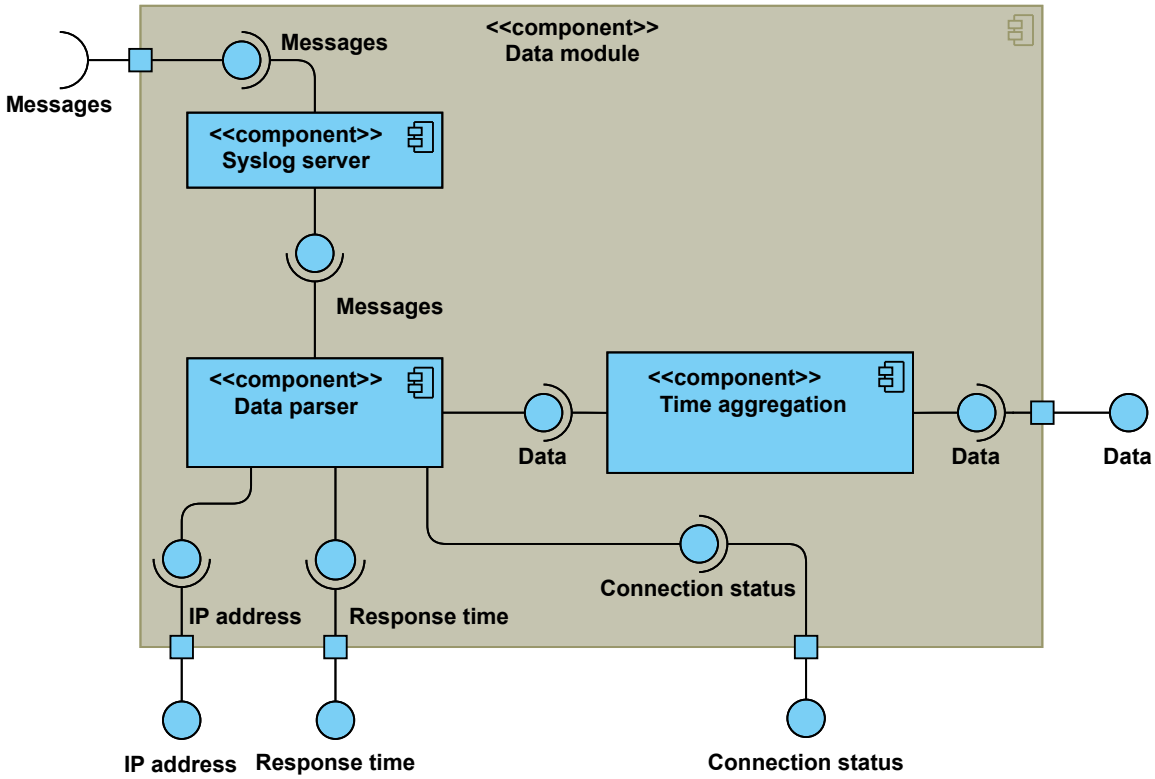

Figure 3.4: Diagram of the Data module

Performance monitor module

Performance monitor, as seen in Figure 3.5, looks at the response time and number of active connections. The response latency can fluctuate based on the complexity of requests as well as the number of requests being processed by the webserver.

There are two approaches for the load estimation. The first one being static baselines, which can be set based on the webserver capabilities or desired behaviour. This can serve as a fail-safe in case of attacks that aim to drain the server resources over a long period of time. The second one is a probabilistic approach that evaluates the current latency and number of active connections in contrast to the past. Both of these attributes should be evaluated separately and the result should be based on the highest severity. This can be then used by the mitigation module for decision-making.

For this evaluation, the requests need to be aggregated into decision time-windows representing the workload over a short period of time. The final size of the time window should be decided based on testing, since we need to use the smallest window possible that would still provide sufficient information. A decision time window of 10 seconds that would be compared to a control time window of 60 seconds will be assumed during this proposal. Tests based on statical base-lining can use the size of the decision window. However, tests based on statistical evaluation need a control window as a comparison. In case of a sudden congestion of traffic, i.e. a DDoS or a Flash flood, we can expect a sudden increase in server resources consumption. This can be best illustrated by a normal distribution. We can calculate the standard deviation σ using Equation 3.1 and the mean value μ using Equation 3.2 from the last control window.

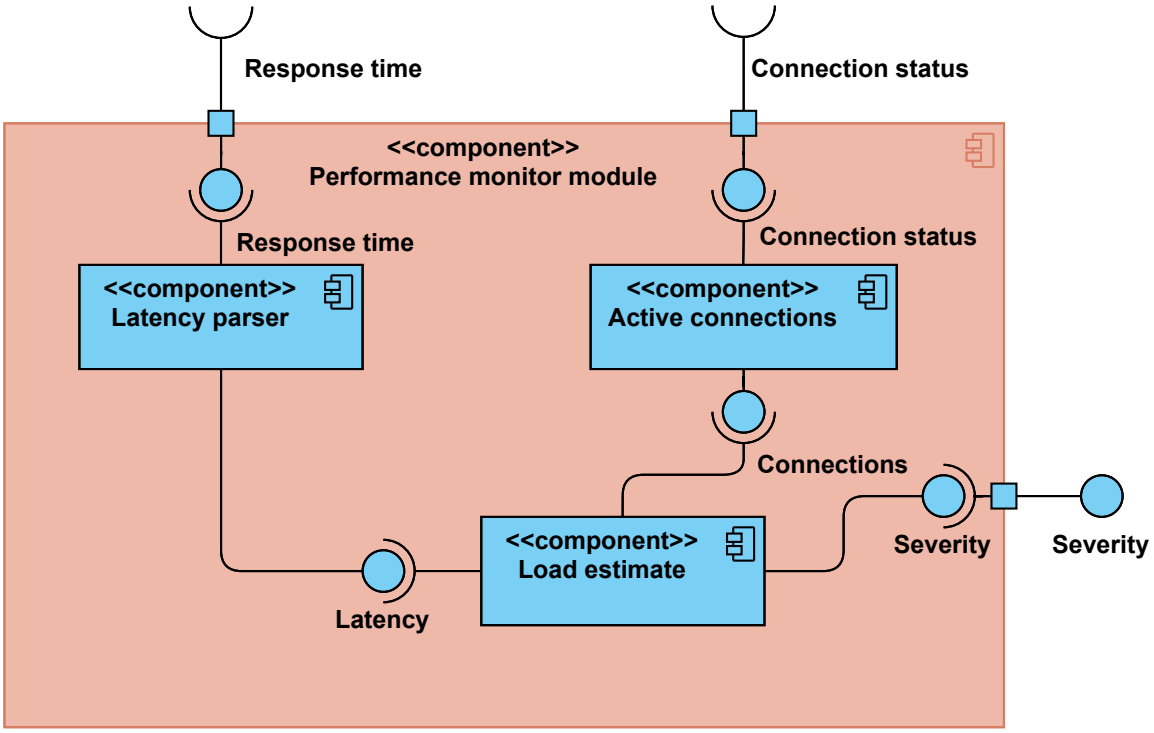

Figure 3.5: Diagram of the proposed Performance monitor module.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}|x_i - \mu|^2} \tag{3.1}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{3.2}$$

The control window will consist of last six decision windows. The change of severity can be based on the distance from the μ. Since we know the μ and γ, we can use this information to decide in which segment of the Empirical rule is the value located. The proposed severity, which is based on the Empirical rule, can be seen in Figure 3.6. This information can then be used to dynamically adjust the severity.

Connection class	Severity
$\mathrm{P}(\mathrm{X} \leq \mu - 2\sigma)$	-3
$\mathrm{P}(\mu - 2\sigma \leq \mathrm{X} \leq \mu - 3\sigma)$	-2
$\mathrm{P}(\mu - 1\sigma \leq \mathrm{X} \leq \mu - 2\sigma)$	-1
$\mathrm{P}(\mu - 1\sigma \leq \mathrm{X} \leq \mu + 1\sigma)$	0
$\mathrm{P}(\mu + 1\sigma \leq \mathrm{X} \leq \mu + 2\sigma)$	$+1$
$\mathrm{P}(\mu + 2\sigma \leq \mathrm{X} \leq \mu + 3\sigma)$	$+2$
$\mathrm{P}(\mu + 3\sigma \leq \mathrm{X})$	$+3$

Figure 3.6: Proposed severity evaluation based on normal distribution.

Attack detection module

The attack detection module consists of five main observed features and their classification. The request rate is further divided into three observed features. The final output of this module is a score of each connection, which should be further interpreted by the mitigation module as depicted in Figure 3.7. Some of the selected features are based on [26], which is described in Chapter 2.3.1. The selection was done in order to get high accuracy and fast detection rates. The features will be evaluated in time windows that should be set based on detection rates. Additionally, while some of the selected features observe the same patterns, the proposed approach for evaluation is different. The goal is to use two time windows: decision time window and control time window, which is larger and consists of several past decision windows. The size of these windows should be refined by testing the model. The features and classification are further described below:

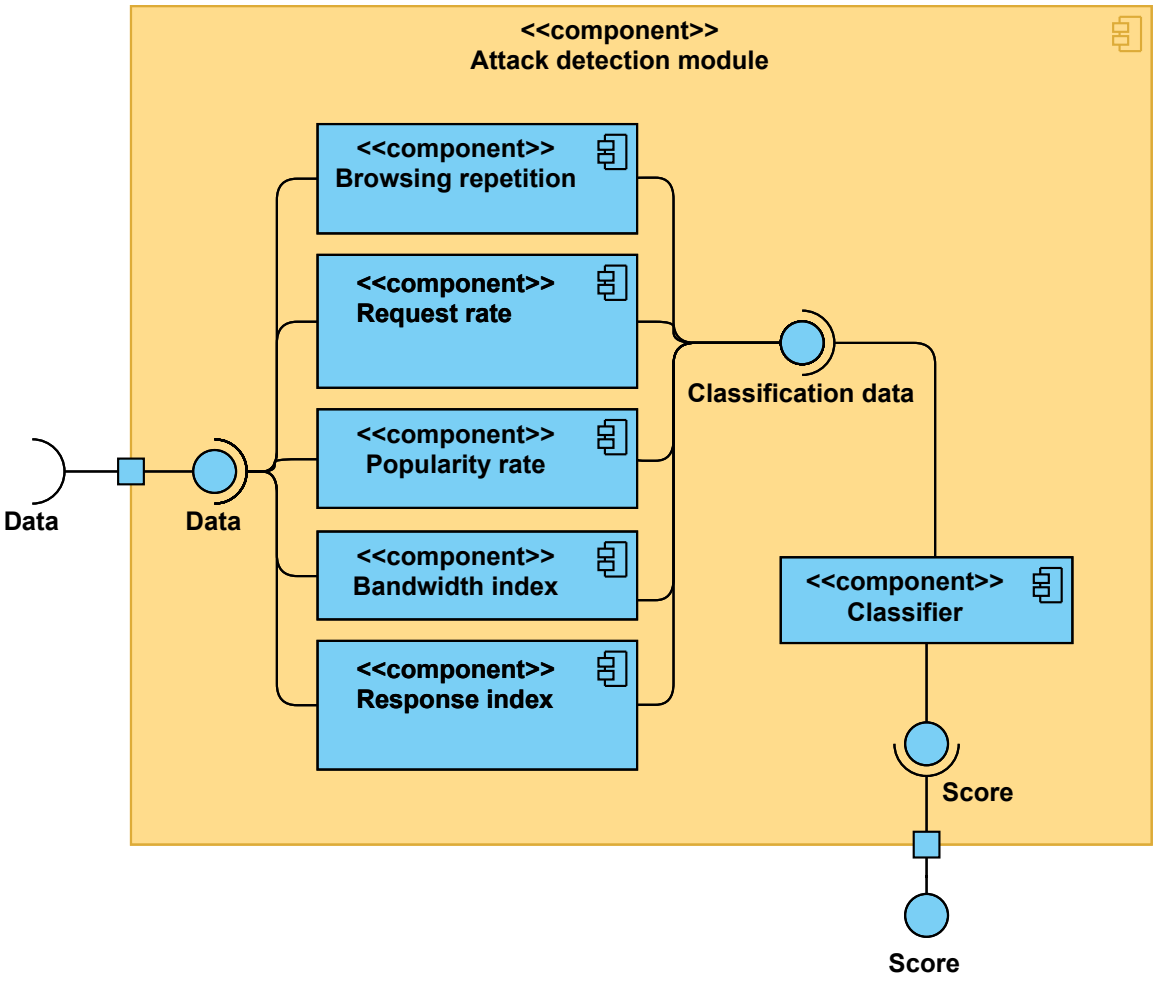

Figure 3.7: Diagram of the proposed Attack detection module.

- **Request evaluation**

  For request rate evaluation, it is possible to use per connection and per IP evaluation. Since it is quite common to see massive network address translation nowadays, for

example in mobile provider networks, the use of per connection vs per IP basis should be decided based on further testing. There can be a problem of bots creating new connections for each request, mimicking the NAT usage.

The first idea is that botnets are not large enough to successfully flood the webserver. The only exception being super-botnets that are much larger and usually perform attacks that can swap the attacking bots and decrease the request rate in order to stay below the detection threshold. Webservers are usually built for their user base; therefore larger webservers can be harder to flood with normal sized botnets. Usually, we can expect increased request rate in contrast to normal user traffic, in some cases even decreased request rate. This can happen either because the botnets are unaware of the actual user traffic or because they are attempting to overload the webserver by gradually increasing their request rate. [26] also proposes that the request rate is affected by the time necessary for the user to process received data.

The second idea is that some botnets might attempt to stay below the rate baselining by keeping more active connections or by holding these connections active below detection rate by keeping pre-set request rates. With legitimate users we can usually expect different behaviour. Thus, an additional observed features consist of the request similarity (RS), which should take into consideration the request rate behaviour between connections as well as the history similarity (HS) that compares history of the connection. The RS computes the difference of decision time windows between current and remaining connections based on Equation 3.3. This value is then divided by all possible connections to obtain the index of similarity.

$$RS = \frac{\sum_{i=1}^{N} identical(i)}{\sum_{i=1}^{N} 1} \tag{3.3}$$

The $identical()$ function finds connections that have identical request rate as the current one.

The HS on the other hand looks at the difference of the same connection. It takes the request rate for individual connections of the current decision time window. These are then compared with all the past decision time windows that are present in the control time window, as we can see in Equation 3.4. The result is the difference between request rates of individual windows. Based on the number of windows that the connection was holding this constant request rate we can then make the evaluation.

$$HS = \max_{\forall i \in Ri} - \min_{\forall i \in Ri} \tag{3.4}$$

The Ri represents an array of request rates for individual connections in time.

Finally, the request evaluation is based on the fact that most users of the same website will have similar, but not same, request rates. This behaviour corresponds to the normal distribution function. Therefore similarly to the Performance monitor module 3.2.2, the aggregated information from the control window can be used to compute the μ and σ. Then the Empirical rule can be applied in order to classify the possibility of such traffic in the decision window. The proposed severity can be seen in Figure 3.8. As we can see, the lower request rates are not interesting in this

case, therefore we can omit the lower bound. This information can be additionally used to observe the request behaviour of the user in time.

Connection class	Severity
$P(X \leq \mu + 1\sigma)$	0
$P(\mu + 1\sigma \leq X \leq \mu + 2\sigma)$	1
$P(\mu + 2\sigma \leq X \leq \mu + 3\sigma)$	2
$P(\mu + 3\sigma \leq X)$	3

Figure 3.8: Proposed request probability severity based on normal distribution.

- **Popularity evaluation**

  Popularity evaluation assumes that most traffic of a website is aimed at a small subset of webpages. This idea is based on [2], who came up with the information that 90% of all traffic requests target only 10% of the websites content. The popularity index based on this information then uses the assumption that botnets can not tell these top pages apart from other pages available on the website. Their behaviour should therefore lead to less top page hits, in contrast to legitimate users. The computation can be seen in Equation 3.5, which was first proposed in [26]. An example can be when a web browser requests additional resources based on the page the client requests. Malicious bots might avoid these resources since they are not interesting for the purpose of the attack.

$$PI = \ln\left(\frac{0.9 \times (1-\delta)}{0.1 \times \delta}\right) \tag{3.5}$$

  δ is computed as a number of hits in the hotpage list vs all page requests for the connection.

- **Response evaluation**

  Response time represents the time it takes for the webserver to process and answer to the user's request. While legitimate users might request some time-consuming operations, usually it is dispersed among the rest of their requests, or is balanced by the number of requests sent to the webserver over the period of time from that specific client. Comparing the response time between the control time window of the webserver and the decision time window of the connection could indicate whether or not the user is attempting to send more time-consuming requests for the webserver than other users. Similarly to the Request evaluation, it is expectable that most response time of users would correspond to the normal distribution. Therefore we can use the Empirical rule and classify according to Figure 3.8.

- **Bandwidth evaluation**

  In order for the botnet to make the server inaccessible for legitimate users, it needs to either send a large number of requests, or overload the server through other means, such as the size of requested information. Therefore, another observed feature should be bandwidth rate, where we compare the user's traffic over a period of time to other users. In this case, we should look at the average size of user requests. Legitimate users will usually send multiple small sized requests before requesting larger objects,

which should disperse the average load of individual users. Malicious bots aiming to drain the server resources are however expected to aim to stay below the request rate threshold therefore minimizing the dispersion of the average bandwidth size. The proposed approach consists of the average size of request in decision window of the connection x and the average size over the control window. The result is then statistically evaluated according to Figure 3.8.

- **Repetition evaluation**

  Last but not least, the browsing repetition feature checks whether the user requested same data multiple times over a short period of time. The idea stems from user's behaviour, where we often see users open multiple webpages, but they rarely open the same page multiple times in a short period of time. In case of bots however, it is possible to open the same webpages in a short period if they are traversing the webpages randomly [26]. Or, more commonly, when the bots are attempting to DDoS the website or make some malicious spam/login attack. Since this information is obtained from the log, we can get the number of unique requests per connection in the decision time window and then divide this by the number of all requests. This approach is depicted in Equation 3.6.

$$REP = 1 - \frac{\sum_{i=1}^{N} unique(i)}{\sum_{i=1}^{N} 1} \tag{3.6}$$

  The $unique()$ function returns 1 for unique requests and 0 if multiple same requests occurred.

  The Empirical Rule can be used to evaluate the results of the repetition to better reflect the current state. The additional advantage of this use-case in contrast to static baselining is that it reflects the websites request pattern.

- **Classification**

  After extracting the features, it is necessary to classify the results. There are several approaches, the simplest being a heuristic analysis, using static baselining that would classify the traffic based on obtained data. More advanced solution would be to use these data in combination with machine learning. This solution however encounters the problem that the machine learning algorithm first needs some flagged traffic, which can be later used for learning. Another solution is to first statistically interpret these data before using a heuristic analysis or machine learning classificator on the statistically evaluated results. The most suitable classification would be to use the statistical evaluation in combination with machine learning classifier. This way, the classifier should be able to spot traffic patterns that are otherwise hard to detect. However before a machine learning approach, it is necessary to first create a suitable classification, which would be able to capture enough results for the basis of the machine learning. The result of the classifier should be the final evaluation of individual connections that will be then used in mitigation module.

  Since the evaluation of the tests already counts on the use of statistical analysis of individual features based on the probability distribution, the implementation will first focus on this part and then prepare the classifier in such a way that it can detect suitable amount of botnet attacks. The first part will consist of heuristical/behavioural

analysis that can be later made more efficient by implementing machine learning strategies, such as the Bayes Regression.

Mitigation module

The mitigation module should ideally consist of challenge and mitigation procedures. As we can see in Figure 3.9, the proposed mitigation module is divided into three main parts, which are described in this subsection.

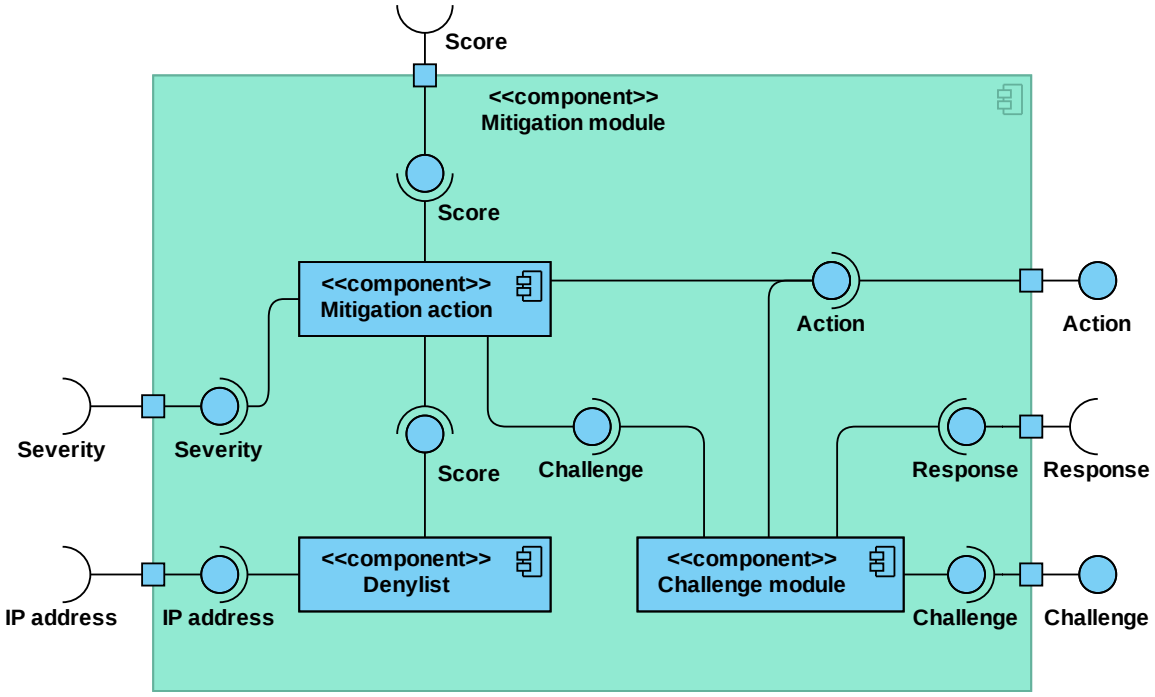

Figure 3.9: Diagram of the proposed Mitigation module.

- **Denylist**

  While the mitigation module aims to classify the traffic according to available data, some addresses might already be denylisted. Therefore, a denylist that can store the IP addresses of attacking networks, date, expiration of the denylisting and also the severity is necessary. This information can later be sent to the proxy to block or slow already detected spurious traffic.

  Additionally, some internet service providers publish lists of IP addresses that are often used in botnet attacks. These lists can be also obtained and added into the denylist to prevent the attack before it occurs.

- **Challenging module**

  The challenging module should ideally consist of four steps, as was described in Figure 2.3.2. The idea is to gradually test bot generations until it is clear the user is not a bot. If the challenging determines that the user is a bot, we can then denylist its IP address for a period of time based on the re-occurrence of the attack as well as the severity of the attack. Other approach would be to use paid extension

such as Google Captcha that could also verify uncertain connections. However, this module will not be implemented because of the difficulty to implement and appropriately verify and test this system. Therefore, this module is added into the proposal of the system and the system can be later extended for limiting the number of false positives while increasing the detection score. This can be achieved since the used classifiers can work in a more strict environment.

- **Decision-making**

  The main part of the mitigation module is the decision-making. This part should take all the available information from the denylist, performance monitor and attack detection module in order to determine whether the user is a bot. This module should work mainly on dynamic thresholding, combining the evaluations from the other modules and dynamically changing the thresholds, updating the deny list and sending the updated information to the proxy for mitigation.

Chapter 4

Implementation

The implementation aims to evaluate webserver traffic in realtime as well as retrospectively. It is divided into two parts. The first part is the DMS that evaluates the incoming traffic on syslog server, upon which the system makes decisions. The second part consists of the use of a receiving module that runs separately from the Nginx proxy/webserver implementation and also necessary configuration of the proxy. This schema can also be modified for Apache webservers. The Nginx webserver was selected based on the theoretical part of this thesis. As was mentioned in 2.1.2, the Nginx implementation is becoming more popular and also offers more possibilities for extension in the future, such as user challenge described in section 2.2.3.

4.1 Communication

The communication between the system parts can be seen in Sequence diagram 4.1. The Nginx proxy sends logs of incoming traffic to the server running the DMS through syslog. Thanks to this division we can spread the load of the traffic between multiple separate servers. After the evaluation is complete, the DMS asynchronously sends the current denylist to the denylist module that receives the data through websocket messages. Last but not least, the denylist module informs the Nginx `http_access_module` about the change in denylist.

4.2 Decision-making server

This section describes the implementation of the decision-making server. The first subsection describes the individual classes of the system and what each part of the system does. The second subsection takes a closer look at the classification steps. Lastly, the final subsection explains the parallelism of the system.

4.2.1 System implementation

The implementation is divided into several classes as can be seen in Attachment A.1. In this subsection, we will describe the classes and logical functions.

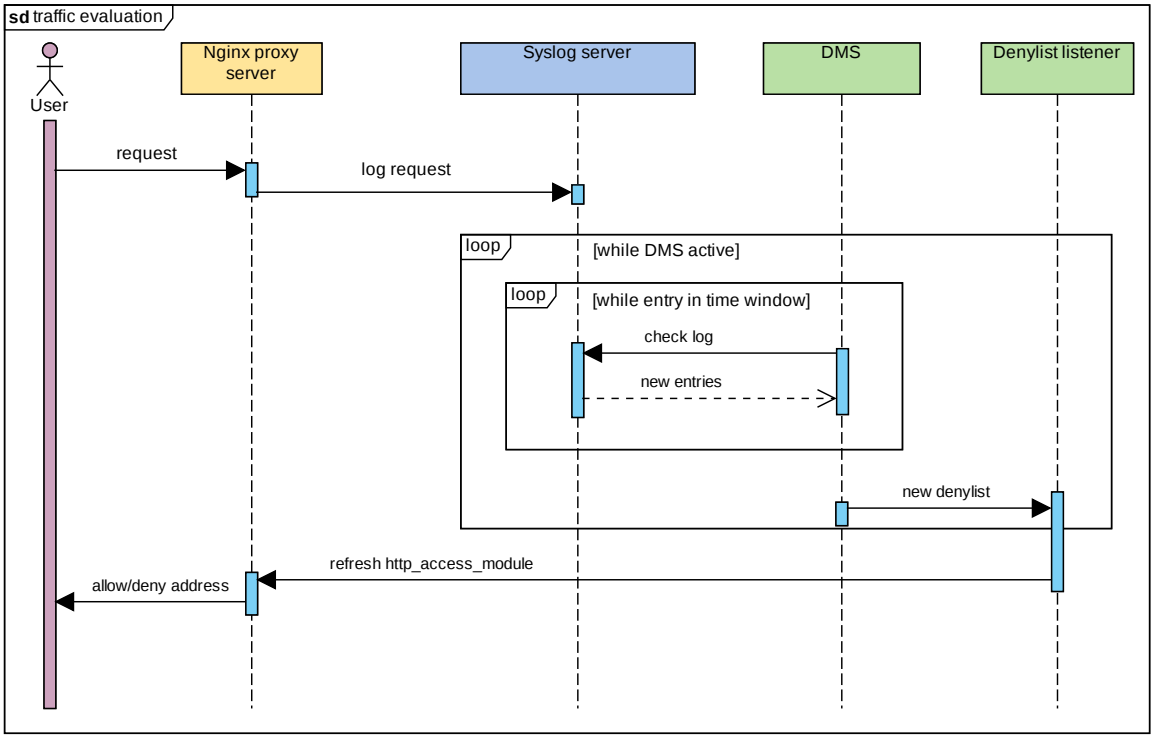

Figure 4.1: Sequence diagram showing the communication between system parts.

Parser

The main class is the parser, which continuously checks the log obtained from the syslog server and stores the requests into the `FeaturesControl` and `PerformanceControl` classes. The class consists of two logical functions.

The first one is the `parsing` function, which contains the main parsing loop monitoring the logs. The parsing consists of reading the log file and creating `Request` objects representing each received request before evaluation. This function works either in realtime or in parsing mode. In realtime, the system continuously observes changes in the log. Additionally, it verifies that no window rotation occurred by checking the timestamps of the requests as well as the system time. This is essential especially for the performance module, which updates the denylist database based on these rotation events. In the parsing mode, the system evaluates traffic from already created logs. The parsing method can be changed in the configuration file.

Function `logHandler` checks if the log is still available and if log rotation occurred. If the system runs in realtime mode, the `logHandler` checks whether log rotation occurred. This is done by checking the inode of the file. If the log was rotated, the function closes the previous file descripor and opens a new one with the current log file. Otherwise, if the log was deleted, it ends the parsing cycle. In the parsing mode, it ensures that file descriptor is correctly closed.

Request

The `Request` class is a data class that holds all data of one request before it is evaluated. This class is filled in the `Parser` class and then distributed to the `PerformanceControl` and

`FeaturesControl` respectively. These classes then insert the `Request` to the corresponding decision class for evaluation.

Control

The `Control` class serves as a template parent class for the `PerformanceControl` and `FeaturesControl` classes. This class contains the window rotation logic as well as the parallelism handling of the system. The class contains three logical functions, namely `copy`, `insert` and `rotate`.

The `copy` function creates a deep copy of the `Control` class that is obtainable by another process. Therefore, the copied class contains all the data of the last decision time window as well as the control time window, which is then evaluated by another process.

Function `insert` inserts new data into the current decision class. Furthermore, it verifies the request time and window time and invokes the window rotation if the request should be already inserted into another time window.

`rotate` function is responsible for the window rotation. The control classes are each instantiated once at the beginning of the system and are accessible by all processes. The window rotation consists of the past decision windows that are instantiated and destroyed based on the size of the respective window. Therefore, it is not necessary to continuously fill the control window, since it consists of the previous decision windows. After the rotation is complete, the function allows another process to call the `copy` function to store the obtained data in the control as well as the decision window for evaluation.

PerformanceDecision

This class holds the data responsible for evaluation of the webserver load. The class has one logical function, namely `insert`, which takes the inserted request and stores necessary data into the class.

PerformanceControl

`PerformanceControl` is a class that extends the `Control` class and is responsible for the evaluation of the webserver load. The class consists of two logical functions, namely `getResponse` and `getDecision`. Both of these functions then gather the data stored in `PerformanceDecision` classes, which are then evaluated based on the normal distribution, as described in Section 3.2.2. These two functions return the evaluation of these two features, which are used for the configuration of `Denylist` during decision-making. The number of decision windows as well as the size of the windows can be set in the configuration file.

FeaturesDecision

Similarly to `PerformanceDecision`, this class holds data aggregated in specified time window. Furthermore, this class also contains the function `insert`, which stores the `Request` object and verifies, if the `user_agent` of the request is accepted. Finally, the class has a logical function `calculate` that makes the necessary aggregation over all requests that were inserted into this class. The aggregation is done by the evaluation process, thus limiting the impact on the parser. The evaluation is done once for each copy and the data are then prepared for evaluation in the `FeaturesControl` functions.

FeaturesControl

This class holds most of the classification logic. The class consists of nine logical functions that represent the main part of the evaluation. Other than that, this class extends the parent `Control` class and uses the `FeaturesDecision` classes that hold the necessary data for the evaluation. Each `FeaturesDecision` class represents one time window. The number of decision windows as well as the size of the decision window can be set in the configuration file. The functions were proposed in 3.2.2 in the Attack detection module part.

Function `getSimilarity` evaluates the similarity of individual connections in current decision window. Specifically, it looks at the number of requests of individual connections and compares it to other connections. The result is the percentage of connections with the same request rate.

The `getHistory` function evaluates the similarity of the connection between different time windows. For each connection, it takes the number of requests and compares it to the other time windows. Additionally, the request rate has to be detectable in at least three time windows. It returns the maximal request rate difference between the time windows.

The `getRequest` function checks the request rates of individual connections and statistically evaluates the results as was described in Section 3.2.2. The function returns an evaluation representing the likelihood of occurrence.

`getPopularity` looks at the number of hotpage hits in the decision window and then statistically evaluates these results in contrast to the control window as proposed in Section 3.2.2. If no hotpage hits are present, the function returns infinity.

The `getResponse` evaluates the response time of the connection. The function compares the individual connections in decision window in contrast to the average response time of the connections in control window. The function then returns an evaluation based on the likelihood of such occurrence as described in Section 3.2.2.

The `getBandwidth` statistically evaluates the request size of the connection in contrast to the average size in the control window. This is done similarly to the `getResponse`.

The `getRepetition` calculates the repetition index of individual connections in decision window, which is then statistically compared to the control window using the Empirical rule as proposed in Section 3.2.2.

Finally, the `classify` function interprets all the results from the individual statistical functions and returns an aggregated evaluation for each connection. This evaluation as well as the timestamp of the occurrence is then stored into the `Denylist` if suspicious behaviour was detected.

Denylist

Class `Denylist` holds the current denylist, which is held in memory and also stored in file in case of system failure. The class is also used in multi-process environment. The process procuring the `features` functionality feeds new addresses into the class, while the process `performance` queries the `Denylist` based on specific situation. The classs consists of five logical functions.

The `load` function works as an init function when the system starts. It loads the denylist file from disk into memory and prepares the handler for file manipulation.

Function `insert` inserts new address into both the cached denylist as well as the file. If the address was already present in the denylist, it can update the timestamp stored in file based on last occurrence.

Finally, the `getDeny` function queries the stored denylist and obtains the denylist based on the specified configuration.

4.2.2 Classifier

The classification is distributed across the system as was described in Section 4.2.1. After the logs are captured, they are aggregated into the time windows. Each time window represents the traffic during this time. The classification consists of two logical windows, namely the decision and control window, which are further evaluated by two different parts.

The first part, the performance evaluation, observes the load on the webserver. This evaluation is based on averaging the time windows and observing if the load continues according to normal distribution as proposed in Section 3.2.2. Sudden increase in traffic that is often generated either by botnets or flash crowds will therefore be detected. This in turn increases the severity of the evaluation, flagging more traffic.

The classification continues by feature evaluation. It consists of 7 observed features as was proposed in Section 3.2.2. Request, repetition, response and bandwidth features are prepared based on the proposal before applying the Empirical rule. The resulting evaluation corresponds to the Figure 3.8. History feature evaluates the connection based on the stability of request rate over time as proposed in Section 3.2.2. Similarity feature measures the closeness of connection request rates. Lastly, Popularity evaluates the number of hotpage hits as depicted in Section 3.2.2. Implementation of those features is further described in Section 4.2.1. Evaluation from each feature serves as an input for the behavioural analysis of the traffic. The behavioural analysis examines the traffic and values of individual features during different traffic patterns. This was used to derive appropriate configuration of the classifier for detecting suspicious patterns. The evaluation usually uses more than one feature in order to limit the number of false positives. This way, multiple types of attacks can be captured. Finally, the suspicious traffic is stored in the denylist.

Based on the performance evaluation, the system pools the denylist and sends the data to the Nginx proxy, mitigating suspicious traffic. This approach seems to have sufficient success rate for both false positives and false negatives. Further analysis of the classification is in Section 5.

4.2.3 Parallelism

The DMS consists of three processes. The parser is single-threaded and stores data for the other processes, which are informed about the availability of data when it occurs. When the data is available, the other processes copy the data and inform the parser to continue. The other processes then evaluate the data increasing the effectivity of the whole system. Features analyser stores data into the denylist class, which is then obtained by the Performance analyser. The simplified communication can be seen in Sequence Diagram 4.2.

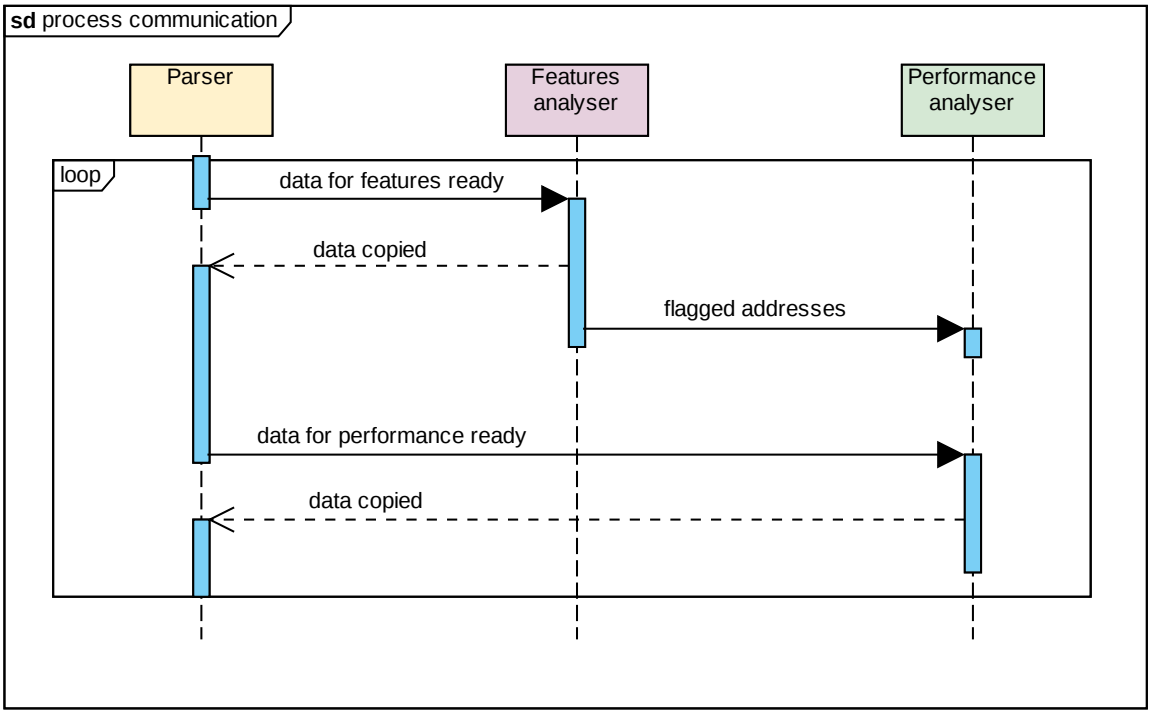

Figure 4.2: Sequence diagram showing the process communication.

4.3 Nginx proxy

The implementation on the Nginx part consists of the configuration, log handling, and program which updates the denylist configuration of the Nginx. In order to successfully setup the Nginx proxy, it is first necessary to install Nginx. The configuration in `/etc/nginx/nginx.conf` should contain line `include /etc/nginx/sites-enabled/*` under the `http` clause. After that it is necessary to add a config file into the folder `/etc/nginx/sites-enabled/` in the format shown in Figure 4.3. This prepares the configuration to run in the Proxy mode. Finally a configuration reload is necessary by running `systemctl restart nginx` command.

```
server {
     listen 80;
     listen [::]:80;
     access_log
          syslog:server=(dms_ip):(port),
          facility=local7,tag=nginx,severity=info
          upstream_rp;
     location / {
          proxy_pass (webserver_ip);
          proxy_bind $server_addr;
          sub_filter "(webserver_ip)" "(proxy_ip)";
          sub_filter_once off;
          sub_filter_types ;
     }
}
```

Figure 4.3: Nginx proxy configuration.

4.3.1 Log handling

The logs are sent by the proxy through syslog messages to the server, which is running the instance of the DMS. The syslog configuration is already prepared in the Figure 4.3. The DMS is constantly observing this log and when new request arrives, it parses the request and stores it into the `PerformanceControl` and `FeaturesControl` classes for evaluation. Additionally, the log needs to be in a specific format, as shown in Figure 4.4, to obtain all necessary information. This log configuration has to be set in the `/etc/nginx/nginx.conf` file under the `http` clause. The format is also appended with the program.

```
log_format upstream_rp
     'addr="$remote_addr" '
     'port="$remote_port" '
     'time="$time_local" '
     'status="$status" '
     'req_size="$body_bytes_sent" '
     'conn="$connection" '
     'connreq="$connection_requests" '
     'req_time="$request_time" '
     'conn_time="$upstream_connect_time" '
     'res_time="$upstream_response_time" '
     'act="$connections_active" '
     'host="$host" '
     'req="$request" '
     'ua="$http_user_agent" ';
```

Figure 4.4: Log configuration for Nginx proxy.

4.3.2 Denylist manipulation

The denylist consists of the Nginx `http_access_module` as well as a support `cpp` program. The program listens on a websocket multicast for the denylist, which is published by the DMS when ready. The program then creates a new denylist file containing the set of received addresses. This list is already cleared of allowlisted addresses. Last but not least, the program forces the Nginx module to refresh, updating the denylist configuration. Before the program can be used, it is necessary to first create the denylist file using `touch (path_to_denylist)`. The next step is to prepare the `http_access_module` by adding `include (path_to_denylist)` to the `/etc/nginx/nginx.conf` under the `http` clause. Now that the configuration is prepared, the module can be started.

Chapter 5

Testing and validation

This chapter presents the testing phase of the system. In the first section, the log analysis is introduced. The second section describes the detection mechanism and the observed reactions of individual features on the traffic. Furthermore, the results are presented as well as possible use-cases and improvements. The last section proposes the system limits based on the observed behaviour during testing.

5.1 Log analysis

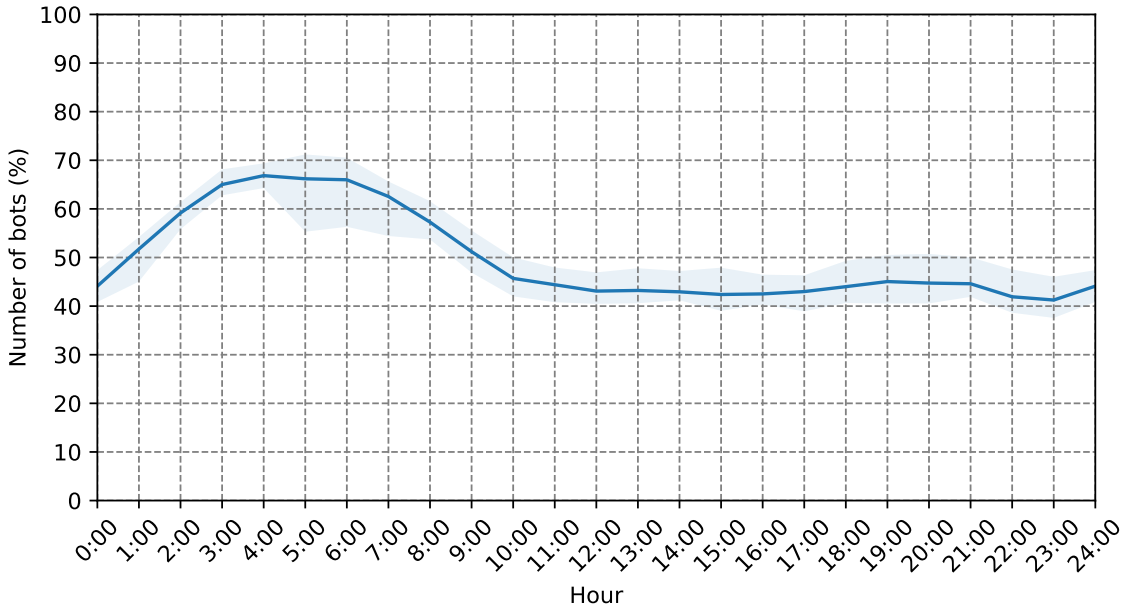

Figure 5.1: Graph showing the average number of bots throughout the day.

The testing phase consists of logs provided by Master Internet s.r.o that were captured between 15th of June and 21st of June. The logs were analysed based on user agents and then remaining addresses with sufficient request rates were verified through IP abuse database. Finally, false positives were manually checked and approximately 60% were also

deemed malicious. This might have skewed the results, since additional addresses were added to the total number that were found in the database[1].

In total, over 956 addresses that used legitimate user agents were flagged as abuse addresses partaking in a botnet or other type of malicious behaviour. Other than that, the traffic consisted of more than 5114 addresses that reported bot traffic in their user agent. Graph 5.1 contains the distribution of bot traffic in obtained logs. The blue line represents the average number of bots throughout the day and the highlighted area represents the maximal and minimal range in which the bot traffic was detected.

As expected, the highest percentage of bots is distributed during the night. This is expectable behaviour since the observed webservers are focused on European market. This distribution can however also skew the evaluation of traffic. Bots nowadays are usually accessing the websites for shorter periods of time or making a few requests and then going dormant in order to not be detected by intervention systems. They can however be replaced by another bot, which continues with the same traffic patterns. For this reason, the evaluation skips the addresses of bots that self-report as bots. This way, we can lower the impact of bots on the statistical evaluation of traffic and obtain the baseline based on legitimate users. The behaviour of unidentified bots is then more visible.

5.2 Detection

The implementation allows the administrator to select trusted bots by setting their user-agent or by adding their IP addresses into allowlist. It also allows the administrator to bypass the detection and block selected user-agents that have malicious behaviour, such as Python scrapers that are often used by competition in order to gain an advantage.

Since the evaluation of bots that are self-reporting as bots is simple, this thesis aims to detect bots that do not self-report. The system works both in realtime where it is able to detect bots as they appear and in analysis mode, where it can go through already captured logs and evaluate the captured traffic. During the testing phase the system found several botnets from Asia that were attempting to verify leaked credentials on specific domains or botnets that were periodically accessing front pages of websites. Other than that, bots that were scraping the websites or attempting to spam in forms and even attempts at XSS scripting were also detected.

The detection of these attacks can be divided into patterns obtained in Table 5.2.

DDoS	constant rate, repetition rate, request rate, similarity, response, bandwidth
Scripting	constant rate, request rate
Login	repetition rate, constant rate, similarity
Scraping	request rate, popularity index

Figure 5.2: Scope of features on different types of attack.

Testing of individual features obtained from the logs is summarized in the following part:

- In terms of request rate, the system mainly focuses on over-the-limit attacks. This is because legitimate users usually make many requests in a short period of time and

[1]Link to the used abuse database: https://www.abuseipdb.com/

then go dormant, or keep low request rates throughout their visit. DDoS attempts to hold the high request rate throughout multiple time windows.

- Repetition of user requests turns out to be a good way of observing the bot traffic. Many bots that focus on Spam, Login and DDoS attacks can be easily detected even at lower request rates. That is mostly because they aim to stay below the request rate threshold and therefore they rather send multiple similar requests and go silent. That is the moment when another bot might take their place and continue the attack. Using the statistical evaluation, it is possible to compare the average number of repetitions for the website or even whole webserver and capture these bots.
- However, the system is able to detect even some low rolling attacks, such as constant rate, that can be detected based on the size of decision window. The tested configuration was able to capture bots with constant rate above 15 requests per minute.
- The similarity rate was able to capture bots at minimal requests per minute, however there were also many false positives. Thus, in the determination phase, the test is modified to capture above-the-limit attacks that come mostly from DDoS attempts.
- The response and bandwidth tests focus mainly on high request rate attacks as well as constant request rate attacks, since they can further decide if the client is attempting to access primarily resources that are time-consuming.
- Finally, the popularity index was not thoroughly tested, because the available logs did not have sufficient request rates to effectively test and evaluate this method. However at large enough request rates, this test should be able to detect scraping bots that do not self-report.

In conclusion, the results of the testing phase can be seen in Graph 5.3 In this graph, we can see the number of not reported bots that were behaving in a way that the system could detect them without high false positive rates.

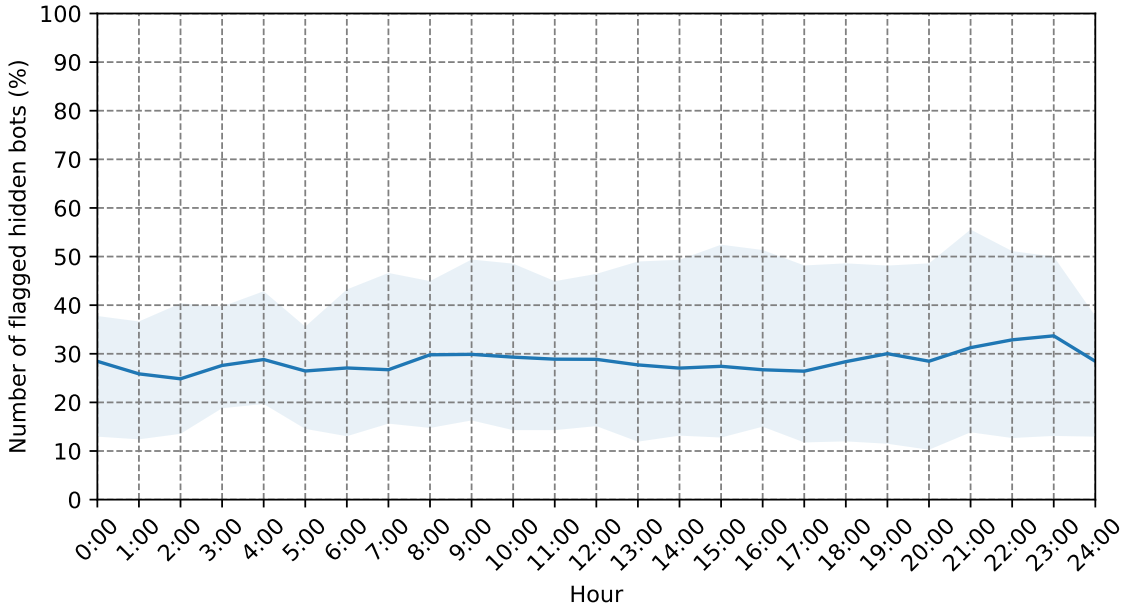

Figure 5.3: Graph showing the average number of flagged bots throughout the day.

The blue line represents the average number of detected bots in a day and their distribution throughout the day. The highlighted area represents the range in which bots were detected throughout the week. As can be seen, the overall capture rate in a day is approximately 30 percent. That is because we focus on above-the-limit bots. Some bots might behave in a way that is almost undetectable or not malicious. Out of all the detected bots, most had similar behaviour to other bots, indicating a possible botnet. Many bots had dynamic rate but sent same requests throughout their presence. The behaviour consisted mostly of main page, login and forms requests. In future work, this detection system could be used to capture the patterns of bots and use machine learning to capture most of the botnets.

By gradually building the denylist database over a large period of time using this tool, the number of bots that are flagged greatly increases as can be seen in Graph 5.4. The blue bar shows the bots that were present in the logs during the day and were gradually captured throughout the week. The orange bar on the other hand shows the bots that were captured in the particular day. Based on this graph, it is obvious that the detection can be greatly increased by building the denylist database first. Furthermore, this graph shows that the detection rate is increasing from day to day. This is however not due to better detection, but rather due to an attack from Asian botnet, which was mentioned before and was composed of approximately 100 addresses. The attack was aimed at login forms and consisted mainly of bots from Vietnam and China.

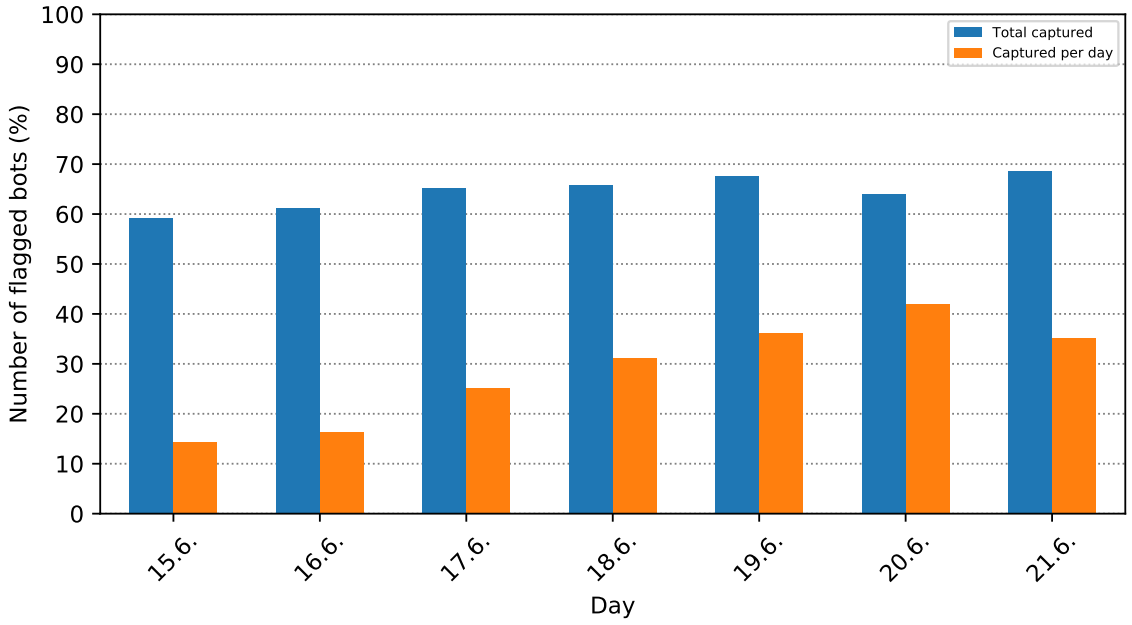

Figure 5.4: Graph showing the number of captured bots per day.

5.2.1 False positives

The number of false positives is dependent on the number of addresses in the logs. For this reason, the classification was done over the whole webserver log file rather than for specific domains. Furthermore, the classification was set on per IP basis. This was done because legitimate users tend to send multiple requests through same connection, while

bots usually use new connection for each request. In this scenario, NAT was not an issue because the traffic was distributed across multiple websites. The number of false positives can be seen in Graph 5.5. As shown in the graph, the number of false positives is below 5%. However, some of these false positives contain traffic such as domain error requests, main page only, or multiple 301 responses. This could indicate web scraping but not enough requests are present for determination.

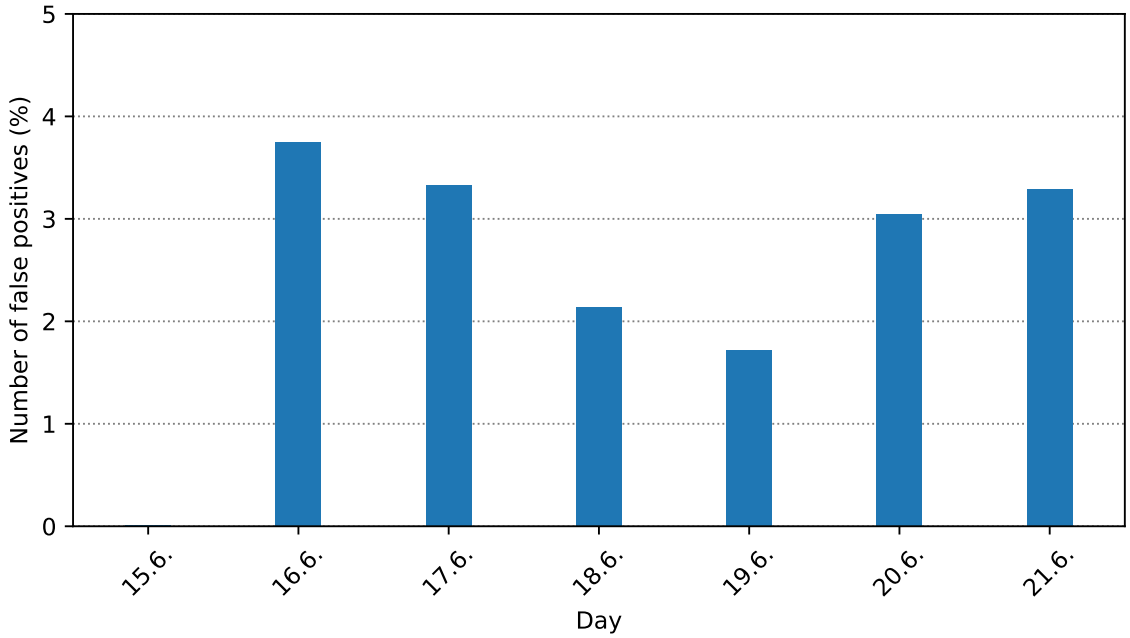

Figure 5.5: Graph showing the number false positives.

5.2.2 Realtime test

During realtime testing of the system, the system was tested in sandbox with the configuration of 20 second decision windows and 4 control windows. The system was able to identify DDoS attacks using HOIC[2], which is one of the common tools [9] used in DDoS attacks. Table 5.6 shows the reaction time of the system for different types of simulated DDoS attacks. In real traffic these statistics can be potentially hidden by the overall request rates of user traffic (additional GET requests for external files, images), which could hide dynamic rate DDoS attacks. These are however not common, since DDoS tools tend to send constant rate.

[2]Link to the HOIC: https://sourceforge.net/projects/highorbitioncannon/

300 requests/minute, static page, constant rate	20s
245 requests/minute, dynamic page, dynamic rate	20-40s
150 requests/minute, static page, constant rate	20s
113 requests/minute, dynamic page, dynamic rate	20-40s
18 requests/minute, static page, constant rate	20s
18 requests/minute, dynamic page, constant rate	60s
20 requests/minute, dynamic page, dynamic rate	undetected

Figure 5.6: Capture time on simulated DDoS.

As we can see, the system is able to quickly detect most DDoS attacks when they occur. Additionally, it is able to capture some low rolling constant rate attacks, which could potentially be attacks such as Slowloris, which aims to starve the webserver resources by keeping multiple connections alive. This however requires the attacker to send requests to keep the connections alive. Many botnets in the real traffic logs however behave in such a way that they send between 10-30 requests and go offline, which is also a behaviour similar to many real users.

Finally, in realtime mode, the system also dynamically changes the severity of the evaluated traffic based on the change in webserver load. This dynamically changes the pooled denylist by severity as well as the time of occurrence.

5.3 System limits

During the testing phase, the system was able to parse 1 million requests in under 8 minutes. In theory this means that the system should be able to process at least 125 thousand requests per minute. If the number of requests exceeds this limit, the system skips these requests until the next time window, where it begins the next evaluation. For comparison, the log containing all traffic from the webserver with multiple websites consisted on average of 1 thousand requests per minute. Therefore, for larger webservers we can split the log for individual websites and run multiple instances of this system to bypass the system limitations.

Similarly, the system can have problems detecting bots if the overall webserver traffic contains under 200 requests/minute. In this case, only constant rate bots are detectable since there is not enough traffic for the statistical evaluation.

Chapter 6

Conclusion

The goal of this thesis was to propose and implement a botnet detection system that would be able to work in realtime. The system is based on time windows that represent the behaviour of traffic over a specified period of time. The evaluation of the time windows is based on statistical approach. The data are first aggregated into the time windows and then individual features are evaluated. Finally, a behavioural analysis is used in order to classify the users.

As proposed in Chapter 3, the system is split into two main parts. The first part is the proxy side, which consists of the log capture, and denylist manipulation. The second part is the main system, which can run separately from the Nginx proxy. The main purpose of the system is to detect traffic that can be attributed to individual bots as well as botnets. This is done by evaluating logs from the traffic and observing the behaviour patterns of each user. The system is capable of running in realtime as well as in parsing mode, where it evaluates already captured traffic.

Based on the testing phase, the goals of this thesis were successful since the system successfully captured botnet traffic and was also able to mitigate such attacks in realtime. During testing of the system, it captured approximately 60% of detected bot traffic that were not self reporting by user-agents. The success rate of the detection is dependent on the time the denylist has to build itself as well as the number of requests present during the time windows. The observed number of false positives was under 5%. During the testing phase, many bots with similar behaviour that were focusing on login attacks and spamming were reoccurring during the week. The limit of the detection system is approximately 125 thousand requests per minute.

In the future, the system can be further extended with several additional functionalities. First, a challenging module can be implemented as described in Chapter 2. This would greatly limit the number of false positives and therefore the system could be configured in a way where it would detect more bots overall. Additionally, mitigation techniques for the system can be expanded. This would decrease the severity of wrongly classified traffic. Finally, the classification can also be extended by machine-learning techniques where the results from this system can be used as the input data for learning.

Bibliography

[1] Akamai. *What is a botnet attack?* Accessed: 2021-02-01. Available at: https://www.akamai.com/us/en/resources/what-is-a-botnet.jsp.

[2] Chengxu Ye and Kesong Zheng. Detection of application layer distributed denial of service. In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. 2011, vol. 1, p. 310–314. DOI: 10.1109/ICCSNT.2011.6181964.

[3] Cloudflare. *What Is A Reverse Proxy?* Accessed: 2020-12-29. Available at: https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/.

[4] Cloudflare. *What is the Mirai Botnet?* Accessed: 2021-02-02. Available at: https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/.

[5] Datadome. *Bot detection: how to identify and block bot traffic to your websites, mobile apps, and APIs*. Accessed 2020-11-02. Available at: https://datadome.co/bot-management-protection/bot-detection-how-to-identify-bot-traffic-to-your-website/.

[6] Datadome. *Web scraping protection: How to protect your website against crawler and scraper bots*. Accessed: 2021-01-29. Available at: https://datadome.co/bot-management-protection/scraper-crawler-bots-how-to-protect-your-website-against-intensive-scraping/.

[7] Gibb, R. *What is a Web Application?* 2016. Accessed: 2021-01-06. Available at: https://blog.stackpath.com/web-application/.

[8] Imperva. *Botnet DDoS Attacks*. Accessed: 2021-02-05. Available at: https://www.imperva.com/learn/ddos/botnet-ddos/.

[9] Imperva. *High Orbit Ion Cannon (HIOC)*. Accessed: 2021-02-05. Available at: https://www.imperva.com/learn/ddos/high-orbit-ion-cannon/.

[10] Infosimples. *Detect Headless*. 2019. Accessed: 2020-12-11. Available at: https://github.com/infosimples/detect-headless.

[11] Jakubová, V. *Bad bot, botnet a spambot: co způsobují a jak se před nimi chránit?* 2020. Accessed: 2021-02-15. Available at: https://www.master.cz/blog/bad-bot-botnet-spambot-jak-se-chranit/.

[12] Kegel, D. *The C10K problem*. 2019. Accessed: 2020-12-29. Available at: http://kegel.com/c10k.html.

[13] Li, K., Zhou, W., Li, P., Hai, J. and Liu, J. Distinguishing DDoS Attacks from Flash Crowds Using Probability Metrics. In: *2009 Third International Conference on Network and System Security*. 2009, p. 9–17. DOI: 10.1109/NSS.2009.35.

[14] MazeBolt. *Dynamic HTTP Flood*. Accessed: 2021-01-22. Available at: https://kb.mazebolt.com/knowledgebase/dynamic-http-flood/.

[15] Ndibwile, J. D., Govardhan, D., Okada, K. and Kadobayashi, Y. Web Server Protection against Application Layer DDoS Attacks Using Machine Learning and Traffic Authentication. In:. July 2015. DOI: 10.1109/COMPSAC.2015.240.

[16] Nelson, R. *Mitigating DDoS Attacks with NGINX and NGINX Plus*. 2015. Accessed: 2021-01-26. Available at: https://www.nginx.com/blog/mitigating-ddos-attacks-with-nginx-and-nginx-plus/.

[17] Nginx. *HTTP Load Balancing*. Accessed: 2020-12-30. Available at: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

[18] Nginx. *NGINX Reverse Proxy*. Accessed: 2020-12-30. Available at: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/.

[19] Nginx. *NGINX Wiki*. Accessed: 2020-12-29. Available at: https://www.nginx.com/resources/wiki/.

[20] Nginx. *What Is a Reverse Proxy Server?* Accessed: 2020-12-30. Available at: https://www.nginx.com/resources/glossary/reverse-proxy-server/.

[21] Ni, T., Gu, X., Wang, H. and Li, Y. Real-Time Detection of Application-Layer DDoS Attack Using Time Series Analysis. *Journal of Control Science and Engineering*. september 2013, vol. 2013. DOI: 10.1155/2013/821315.

[22] Radware. *IDBA: A Patented Bot Detection Technology*. 2019. Accessed: 2020-10-30. Available at: https://blog.radware.com/security/2019/06/idba-a-patented-bot-detection-technology/.

[23] Radware. *Meet the Four Generations of Bots*. 2019. Accessed: 2020-10-30. Available at: https://blog.radware.com/security/2019/09/meet-the-four-generations-of-bots/.

[24] Saita, A. *Chameleon Botnet Stealing $6M a Month in Fraudulent Ad Clicks*. 2013. Accessed: 2021-02-06. Available at: https://threatpost.com/chameleon-botnet-stealing-6m-month-fraudulent-ad-clicks-032013/77651/.

[25] SARAVANAN, R., SHANMUGANATHAN, S. and Palanichamy, Y. Behavior-based detection of application layer distributed denial of service attacks during ash events. *TURKISH JOURNAL OF ELECTRICAL ENGINEERING & COMPUTER SCIENCES*. january 2016, vol. 24, p. 510–523. DOI: 10.3906/elk-1308-188.

[26] Singh, K., Singh, P. and Saluja, K. User behavior analytics-based classification of application layer HTTP-GET flood attacks. *Journal of Network and Computer Applications*. march 2018, vol. 112. DOI: 10.1016/j.jnca.2018.03.030.

[27] Sree, R. and Saira Bhanu, M. HADM: detection of HTTP GET flooding attacks by using Analytical hierarchical process and Dempster-Shafer theory with MapReduce: Detection of HTTP GET flooding attack by using AHP and DST with MapReduce. *Security and Communication Networks*. october 2016, vol. 9. DOI: 10.1002/sec.1611.

[28] TechTarget. *Web application*. 2019. Accessed: 2021-01-21. Available at: https://searchsoftwarequality.techtarget.com/definition/Web-application-Web-app.

[29] Vastel, A. *Detecting Chrome headless, new techniques*. 2018. Accessed: 2020-12-11. Available at: https://antoinevastel.com/bot%20detection/2018/01/17/detect-chrome-headless-v2.html.

[30] W3Techs. *Historical yearly trends in the usage statistics of web servers*. 2021. Accessed: 2021-01-01. Available at: https://w3techs.com/technologies/history_overview/web_server/ms/y.

[31] Yu, S., Zhou, W., Jia, W., Guo, S., Xiang, Y. et al. Discriminating DDoS Attacks from Flash Crowds Using Flow Correlation Coefficient. *IEEE Transactions on Parallel and Distributed Systems - TPDS*. june 2012, vol. 23. DOI: 10.1109/TPDS.2011.262.

[32] Zargar, S. T., Joshi, J. and Tipper, D. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys Tutorials*. 2013, vol. 15, no. 4, p. 2046–2069. DOI: 10.1109/SURV.2013.031413.00127.

[33] Zhang, H., Taha, A., Trapero, R., Luna, J. and Suri, N. SENTRY: A Novel Approach for Mitigating Application Layer DDoS Threats. In:. August 2016, p. 465–472. DOI: 10.1109/TrustCom.2016.0098.

[34] Zhao, D., Traore, I., Sayed, B., Lu, W., Saad, S. et al. Botnet detection based on traffic behavior analysis and flow intervals. *Computers & Security*. november 2013, vol. 39, p. 2–16. DOI: 10.1016/j.cose.2013.04.007.

Appendix A

Class diagram

Figure A.1: Class diagram of the DMS.