Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Linux Based L7 Firewall

Bachelor's Thesis

2024
Terézia Kabátová

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Linux Based L7 Firewall

Bachelor's Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Computer Science
Supervisor:	RNDr. Jaroslav Janáček, PhD.

Bratislava, 2024
Terézia Kabátová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Terézia Kabátová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Linux Based L7 Firewall
L7 firewall založený na Linuxe

Anotácia: Firewally využívajúce len informácie z linkovej, sieťovej a transportnej vrstvy v súčasnosti často nie sú dostatočne spôsobilé zabrániť rôznym útokom. Preto sa čoraz častejšie využívajú firewally, ktoré využívajú aj informácie z aplikačnej vrstvy (L7). Bakalárska práca sa venuje analýze možností, návrhu a implementácii L7 firewallu na báze OS Linux, ktorý umožní validáciu vybraných aplikačných protokolov a filtrovanie komunikácie aj na základe informácií z aplikačnej vrstvy týchto protokolov.

Cieľ:
- analyzovať možnosti a zvoliť vhodnú množinu aplikačných protokolov pre pokročilú kontrolu
- analyzovať a navrhnúť možnosti pokročilej kontroly zvolených aplikačných protokolov
- navrhnúť a implementovať riešenie
- otestovať riešenie a posúdiť jeho výkonové parametre

Vedúci: RNDr. Jaroslav Janáček, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 09.10.2023

Dátum schválenia: 16.10.2023 doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Terézia Kabátová
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Linux Based L7 Firewall

Annotation: Firewalls that utilize information only from the link, network and transport layers are often not sufficiently capable of preventing various attacks. Firewalls which use information also from the application layer (L7) are more effective, therefore, increasingly used. This bachelor's thesis focuses on the analysis of possibilities, design and implementation of an L7 firewall based on OS Linux, which will provide validation of selected application protocols and filtering of communication based also on information from the application layer of these protocols.

Aim:
- analyze possibilities and choose an appropriate set of application protocols for advanced validation
- analyze and propose advanced validation techniques for the selected application protocols
- design and implement the solution
- test the solution and assess its performance parameters

Supervisor:	RNDr. Jaroslav Janáček, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.

Assigned: 09.10.2023

Approved: 16.10.2023 doc. RNDr. Dana Pardubská, CSc.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: First of all, I would like to thank my supervisor, RNDr. Jaroslav Janáček, PhD., for offering me to work on this topic, all the helpful advice and constructive collaboration throughout the development of the firewall and writing of this thesis. I would also like to thank Lukáš Horňáček for invaluable feedback on the text of this thesis, my parents, who supported me while I worked on this thesis and everyone else who helped and encouraged me.

Abstrakt

Táto práca sa zaoberá možnosťami implementácie L7 firewallu v user-space OS Linux. Väčšina existujúcich riešení je buď príliš drahá pre verejne financované organizácie alebo ich funkčnosť nezodpovedá definícii firewallu. Prvá časť tejto práce popisuje motiváciu, potrebnú terminológiu a stanovuje rozsah implementácie formulovaním požiadaviek a výberom vhodnej množiny aplikačných protokolov, ktoré sa dajú zneužiť pri absencii L7 kontroly. V ďalšej časti sa rozoberá návrh a architektúra implementácie, problémy, ktoré sa vyskytli počas vývoja a spôsob ich riešenia. Súčasťou práce je implementácia firewallu podľa požiadaviek vyplývajúcich z analýzy. V poslednej časti sú uvedené testovacie scenáre a výsledky, ktoré preukazujú funkčnosť firewallu, a dokaldujú ocharnu proti zneužitiu vybraných protokolov aplikačnej vrstvy. Implementácia firewallu a výsledky tejto práce môžu slúžiť v ďalšom skúmaní a vývoji L7 firewallov na open-source platformách.

Kľúčové slová: L7, firewall, Aplikačná vrstva, Linux, user-space

Abstract

This thesis explores the possibilities of implementing an L7 firewall in the user-space of OS Linux. Most existing solutions are either too expensive for publicly funded organizations or their functionality does not fit the definition of a firewall. The first part of this thesis describes the motivation, the necessary terminology, and sets the scope for the implementation by formulating the requirements and choosing a set of Application layer protocols, which can be abused in the absence of L7 inspection. The next part discusses the design and architecture of the implementation, the issues encountered during development and how they were addressed. The thesis includes the implementation of the firewall according to the requirements resulting from the analysis. The last part presents test scenarios and results that demonstrate the functionality of the firewall, and the protection against abuse of the chosen Application layer protocols. The firewall implementation and the results of this thesis can be used in further research and development of L7 firewalls on open-source platforms.

Keywords: L7, firewall, Application layer, Linux, user-space

Contents

List of Figures

List of Tables

Introduction

> „It is important to draw wisdom from many different places. If we take it from only one place, it becomes rigid and stale."
>
> — Uncle Iroh, Avatar the Last Airbender

In the field of cybersecurity, firewalls are not a novel concept. They started appearing with the advent of malicious practices on the Internet as simple filtering devices [1]. Since then, many types of firewalls and related systems have been introduced. The *first chapter* discusses the necessary terminology from the networking field and the individual firewall types that exist. Each type offers a different set of features that enable administrators to filter out unwanted traffic and prevent it from reaching hosts on their networks. In this thesis, we explore the feasibility of implementing a firewall in the user-space of OS Linux with advanced inspection capabilities, specifically L7 or Application layer analysis. The need for L7 analysis is a result of ever more sophisticated attacks, that leverage L7 protocols. In the *second chapter*, we introduce a set of L7 protocols, that our firewall can analyze and how we chose them. For each protocol, we describe a set of associated attacks and evasion techniques, which are effecitive in the absence of an L7 firewall, and analyze possible countermeasures involving Application layer analysis. The *third chapter* establishes firewall implementation requirements. In the *fourth chapter*, we focus on the design and architecture of the implementation. This includes the definition of the individual modules and their functionality, and how they interact and perform the network traffic analysis. During the design process, we considered different approaches. The decision process is described in the third chapter. Besides the implementation, we also discuss the challenges we encountered during the development and how we addressed them. The source code of the finished implementation is included in Appendix A. In the *fifth chapter*, we present the test methodology and the interpretation of the functional and performance test results. The test results are included in Appendix F. The functional test scenarios cover the attacker techniques described in the second chapter and demonstrate how Application layer analysis implemented in our firewall can defend against them.

Chapter 1

Firewalls

In this chapter, we will briefly describe networking layered models and introduce different types of firewalls. We will also informally describe what our firewall should do and how it differs from other existing solutions.

1.1 Models

In the networking field, layered communication models are used to simplify the representation of network communication. We will focus on the OSI and TCP/IP models. Both are necessary for a proper definition of firewall types and associated terminology. These models use hierarchically organized layers, that constitute communication between hosts on the network. Each layer utilizes the functionality of the lower ones and provides a more abstract view of the communication to the higher layers. A *protocol* describes a set of rules, that govern the interactions on a given layer. For example, a protocol on the lowest layer could define, what encoding or modulation technique should be used with a given medium, such as optical fiber or radio waves. We will now describe the two aforementioned models.

OSI model This model defines seven layers. The lowest layer is named *Physical layer*, as it deals with the transmission of information over a medium, and the highest layer is called *Application layer*, which handles transporting user-facing application data. All layers are listed in Figure 1.1. This model is referential, i.e. it defines each layer's responsibilities in the communication process and provides a framework for standardization, but it does not specify any protocols. The layers are usually referred to by numbers, e.g. the lowest, or Physical layer, is L1 and the Application layer is L7.[1]

TCP/IP model This model has only four layers while maintaining all functionality

[1]These two terms will be used interchangeably throughout this thesis.

defined in the OSI model. This is achieved by merging the top three and bottom two layers. Unlike the OSI model, TCP/IP defines a set of concrete protocols that can be implemented. It is widely adopted in practice, as it is the basis for the Internet.

In this thesis, we will be working predominantly with the TCP/IP model. The OSI model will serve as a source of generic terminology.

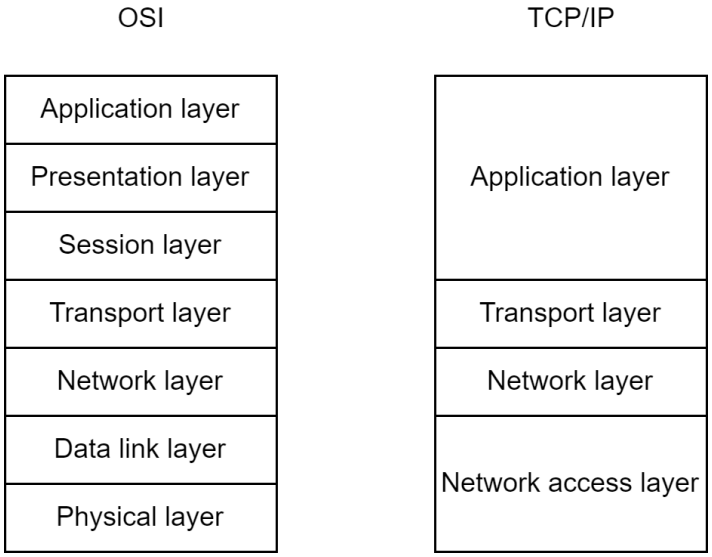

Figure 1.1: Networking models comparison

1.2 Firewall Classification

Firewall can be defined as any system, whether a hardware appliance or purely software, that is capable of filtering network traffic based on data from protocol headers and administrator-defined rules. We will now introduce two types of firewalls, based on which layers they can analyze.

1.2.1 Classic firewall

Classic firewalls can analyze information only from the Internet and Transport layers of the TCP/IP model. An example of an implementation with such filtering capabilities is the Netfilter framework, which is included in the Linux kernel [2]. Information from the Internet and Transport layer protocol headers of a packet is matched against rules and in case of a match, the rule specifies whether to allow or reject the packet. Such a rule could be used for example to prevent SSH traffic from reaching a certain host. The rule would match and drop packets with specific destination IP address, Transport layer

protocol TCP and destination port 22. These rules can be configured from user-space by an administrator using command line utilities, such as iptables or nftables. Classic firewalls are an essential tool for securing networks, but a more sophisticated attacker can easily evade them.

Port spoofing

As we pointed out, classic firewalls utilize information only from the Internet and the Transport layer. We will demonstrate that this scope is not sufficient and the firewall can be easily bypassed. When blocking Application layer protocols using a classic firewall, administrators have to rely on IANA-assigned port numbers [3]. For example, when trying to drop all SSH traffic, the administrator would create a rule that drops TCP segments with port number 22. To evade this, the attacker can use a technique called *port spoofing* — the communicating hosts are reconfigured, so that the port on the Transport layer is different from the officially assigned one. This way an arbitrary Application layer protocol can pass through the firewall. An example scenario could involve a classic firewall configured with a rule that blocks everything, including the standard SSH port, and another rule that allows TCP port 443. This scenario is not unrealistic, as many firewalls forbid incoming communication from the outside except requests destined for servers, which are behind the firewall. In our case, there could be a web server, which accepts HTTPS connections. If the attacker is able to configure the SSH server to use port 443, then they can successfully create an SSH connection and evade the firewall.

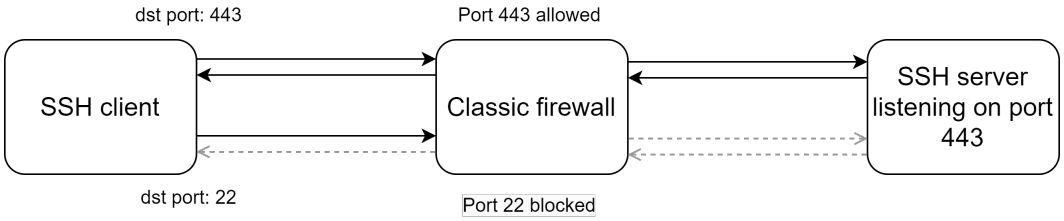

Figure 1.2: SSH port spoofing

The classic firewall is incapable of any type of Application layer analysis, so the SSH traffic remains hidden. In order to detect this type of spoofing, the firewall has to be capable of *Application layer protocol validation*, i.e. checking if the application layer data matches the protocol definition. Then, in combination with the Transport layer inspection, we can enforce policies, such as blocking any TCP traffic with port 443 that does not conform to the HTTPS message format.

1.2.2 L7 firewall

As we have shown, classic firewalls are not sufficient when faced with more sophisticated evasion techniques that leverage the Application layer protocols. Due to this, administrators are unable to effectively enforce security policies. A possible solution to this is employing an *L7 firewall*. Unlike classic firewalls, these are capable of Application layer protocol validation and analysis, i.e. they are capable of including the Application layer data in the decision process. This makes them a suitable measure against the described techniques.

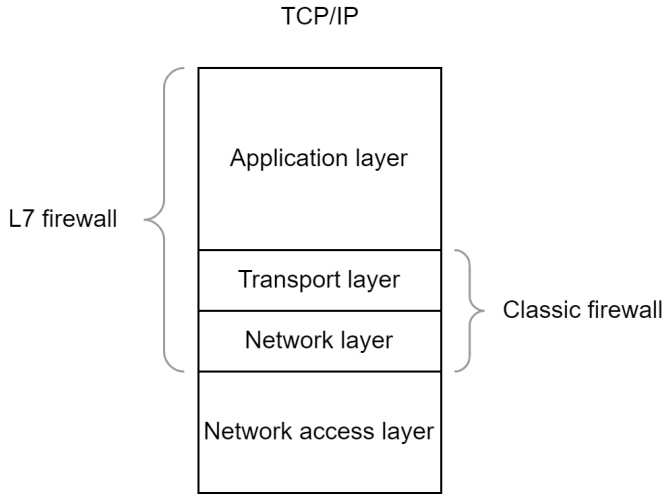

Figure 1.3: Firewalls in context of the TCP/IP model

1.3 Our solution

In this thesis, our goal is to create a solution, which fits the definition of an L7 firewall. That means our software will focus on interpreting Application layer protocol messages and subsequent filtering based on rules, that were configured by an administrator. These rules can specify conditions for the Internet, Transport and Application layers of the network traffic.

1.3.1 Existing solutions

Many vendors and open-source projects already offer solutions with these features. We will now discuss a few examples and how our work differs.

Firstly, we will discuss commercially available products with described capabilities. Many vendors offer not only software, but entire appliances, which act as a firewall with advanced filtering capabilites [4, 5]. These devices are equipped with specialized

components and leverage hardware offloading for some parts of the traffic processing [6]. It enables them to analyze vast amounts of network traffic, and therefore they are suitable for large organizations or enterprises, where high data rates are common. Besides the specialized hardware, other factors, such as an extensive feature set, increase the final cost of the product and make it inaccessible to smaller or publicly funded organizations. Our goal is to create an implementation with L7 firewall capabilities, that does not rely on any specialized hardware and can process traffic in smaller networks, which typically operate with 1 Gbps links.

We will also mention a specialized type of L7 firewalls — *Web application firewalls*. These are available as commercial [7] and open source solutions as well [8]. They can analyze HTTP traffic, which makes them well-suited for preventing attacks on web applications, such as SQL injection, cross-site scripting, etc. In this thesis, we aim to create a more versatile solution, that will be applicable in varying scenarios.

The last solution we will discuss are intrusion detection and prevention systems (IDS/IPS). Examples of such software are Snort [9] and Suricata [10]. They are both characterized as an open source IDS/IPS and differ mainly in implementation [11]. The similarity of an IDS/IPS to an L7 firewall is in the ability to analyze the Application layer data, but its primary purpose is different. IDS/IPS is designed to identify ongoing attack attempts based on suspicious patterns and signatures that are available in the form of curated rule sets.[2] On the other hand, a firewall is used to implement rules that do not necessarily detect malicious traffic but rather enforce a security policy that is defined by an administrator. For example, let's assume an administrator wants to prevent users from connecting to a server via SSH from a specific network. This can be done by configuring a rule on a firewall, which filters the incoming traffic based on source IP address and destination Transport layer port. A user attempting to connect to the server is violating the administrator's policy formulated in the rule, and not executing an attack with a recognizable signature.

[2]For example, the Cisco Talos group publishes a rule set for Snort. Further information about it and IDS/IPS rule sets in general can be found on their FAQ GitHub page in the Rules section — `https://github.com/Cisco-Talos/snort-faq/tree/master`.

Chapter 2

Application layer protocols

In this chapter, we will introduce a set of Application layer protocols that our firewall will be able to recognize, and ways how an attacker may abuse them if the only defense measure is a classic firewall. These protocols were chosen primarily based on their importance for the operation of the Internet and the inability of classic firewalls to detect associated attacks and evasion techniques.

2.1 DNS

The main purpose of the Domain Name System (DNS) is mapping domain names to IP addresses and other information[1] in the form of DNS records. The records are stored in a hierarchical structure of servers. The domain names form a tree structure, where the root domain name is . (a dot). Its direct descendants are *top-level domains*, such as org, com, etc. Domain names on different levels are separated by a dot, e.g. www.example.com are domain names on three levels of the tree. A *zone* is a connected subgraph of the described tree, that is managed by some entity (individual or organization). An *authoritative name server* for a zone is responsible for answering queries regarding the zone. On the other hand, a *recursive name server* accepts queries from hosts and resolves the domain names on their behalf. This is done by traversing the tree structure of domain names until an authoritative name server provides a definitive answer, that is sent back to the host who requested it and cached by the recursive name server for future queries.

The functionality provided by DNS is essential for the operation of the Internet. For this reason, we cannot afford to disable or block it entirely and many attackers take advantage of it [12].

[1] Such as DNSSEC records.

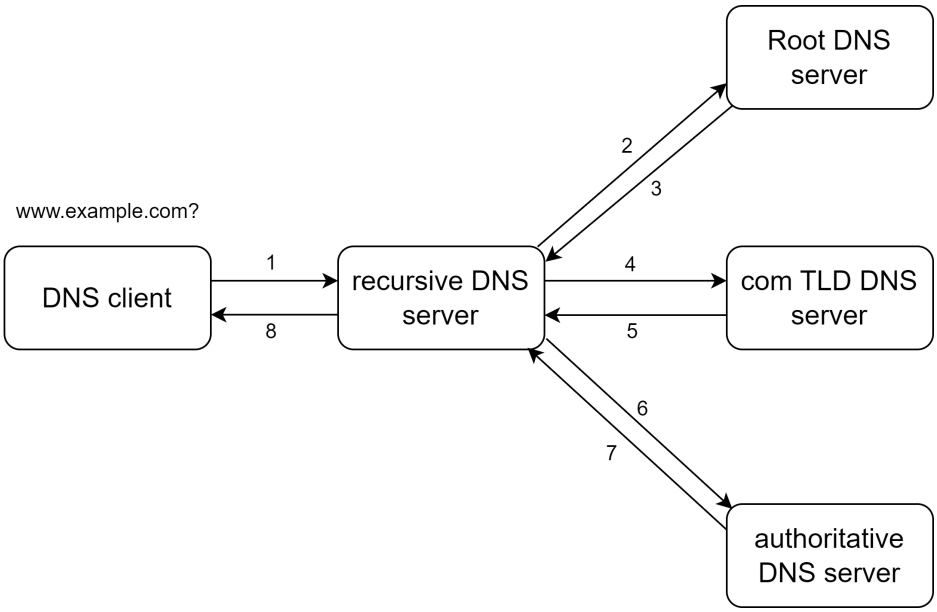

Figure 2.1: Operation of DNS

2.1.1 Tunneling

The port spoofing is a rather simple technique compared to other commonly used practices, many of which cannot be detected by Application layer protocol validation. To illustrate this, we will now examine a technique called *DNS tunneling*. In an example scenario, the attacker is in control of an arbitrary domain and its authoritative DNS server. A compromised host behind a classic firewall can now communicate with the attacker using DNS queries regarding the attacker's domain, where the messages are encoded in the form of subdomains. The host sends its query to a recursive DNS server, which will eventually forward it to the authoritative server owned by the attacker. Now the message can be easily obtained from the query and the attacker can send a reply encoded in a response to the query. This technique can be used for botnet command and control, where the attacker sends instructions via the DNS responses, or data exfiltration, where the data is broken down into small chunks and sent out in the subdomain queries.

This process is practically undetectable by a classic firewall and Application layer protocol validation as it perfectly resembles regular DNS traffic. It is also relatively easy to execute, as tools for creating this type of tunnel are readily available [13, 14]. A feasible defense approach is a more thorough and stateful analysis of the Application layer data. For instance, some tools used for DNS tunneling utilize infrequent DNS record types, so computing and examining the record type frequency is a possible approach when dealing with this technique [15].

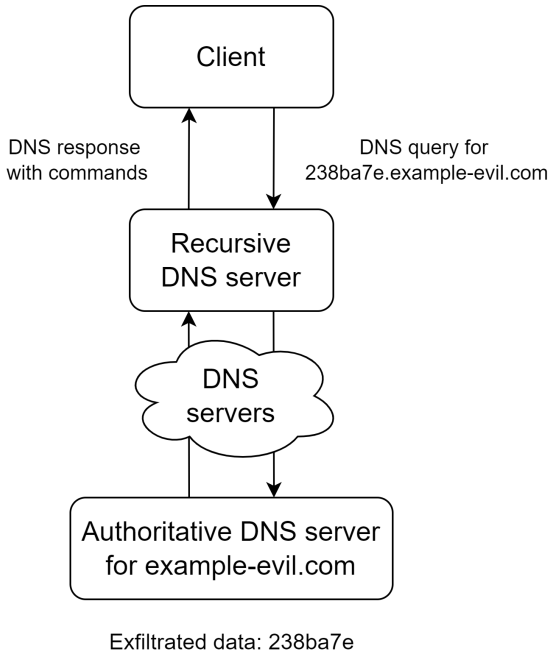

Figure 2.2: DNS tunneling

2.1.2 Reflection and amplification

Another type of attack that abuses DNS is *reflection and amplification*. This attack uses vulnerable open resolvers, i.e. publicly available recursive DNS servers. The attacker sends queries with altered source IP address so that the response will be delivered to the victim. The resolver then sends, or „reflects", the response to the victim. The DNS query is relatively small compared to the response, so the attacker can easily generate large amounts of traffic, that will eventually overwhelm the victim and result in Denial of Service (DoS). This is the amplification aspect of the attack.

A distributed version or DDoS can be executed using a botnet. The bots are the ones sending queries to open resolvers, not the attacker, thus making it exceptionally difficult to find and block the source of malicious traffic.

DDoS attacks are particularly difficult to defend against. It usually requires the use of several systems and techniques. The role of a firewall in this scenario is to rate limit the incoming traffic and thus prevent it from overwhelming hosts on the protected network. The limits are usually not applied to all traffic, but only certain packets, that match conditions. For example, to mitigate an ongoing DDoS attack abusing DNS, the administrator could configure rate limiting for port 53/UDP. Such functionality is implemented in the iptables hashlimit module [16]. However, the conditions are limited to the Network and Transport layers only. We aim to create a similar module for our firewall, which will let the administrator configure conditions regarding Application layer protocol data.

2.2 HTTP

HTTP enables the sharing of hypertext documents and multimedia content. It operates in a client-server architecture and a request-response model. Clients, e.g. web browsers, send requests to web servers, which respond with hypertext documents. The *World Wide Web* (WWW) is the system of hosts on the Internet, that share content via HTTP. In the beginning, it was a simple system for accessing static documents known as *web pages* and other media through a web server, each uniquely identified by a URL. Since then, the WWW has evolved. The web pages became dynamic, i.e. their content is generated based on information included in the request, and then web applications were introduced. A *web application* is an application software, that runs on the side of the web server and the user can access it through a web browser in the form of a web page.

HTTP does not ensure confidentiality, authenticity and integrity of the transferred content, as no encryption scheme or other cryptographic construction is used. HTTPS was introduced as a secure version of the protocol. It transfers HTTP messages through a secure channel provided by TLS. If configured correctly, this eliminates the concerns associated with plain HTTP. Most web servers today support and actively use HTTPS. According to web browser vendor statistics, approximately 90% of the HTTP traffic is encrypted [17, 18]. The encryption prevents a firewall, or any other device on the network, from reading and examining the HTTP messages. Some commercially available firewalls offer *TLS inspection*, where the firewall acts as a proxy and can decrypt the TLS messages and analyze the content [19]. This is however out of scope for this thesis. We will only focus on HTTP, specifically HTTP/1.1. According to statistics, in 2021 HTTP/1.1 accounted for nearly 40% of analyzed webpage loads and port 80 — the IANA assigned port for HTTP — is still one of the most commonly open ports on the Internet [20, 21].

Attacks associated with HTTP can abuse vulnerabilities associated with the web server, such as buffer overflows or inadequate access control. We will now discuss them in detail.

2.2.1 Buffer overflow

The HTTP protocol itself does not impose any limits on the length of the messages or individual fields of the messages. However, web server implementations may use fixed-length buffers to store the incoming data. In this case, the message length is effectively limited, and the implementation may be vulnerable to buffer overflow attacks. Successful exploitation of these vulnerabilities can result in denial of service, reading the memory of the server or malicious code execution. One possible mitigation technique is the enforcement of the length limits using an L7 firewall.

2.2.2 Policy enforcement

Even if the Application layer data matches the HTTP protocol definition, i.e. the protocol validation was successful, the message is not necessarily harmless. For example, an attacker can use valid HTTP requests to perform *URL fuzzing*, i.e. generating requests in an attempt to identify hidden directories or trigger unexpected behavior of the web server. We can address this by defining rules on the firewall, that prohibit such requests. For example, we could drop requests for resources that do not exist on the server or should not be accessible from an outside network.

2.2.3 Port spoofing

As we mentioned, port 80 is one of the most commonly open ports on the Internet and many firewalls explicitly allow it. This makes it a good candidate for the port spoofing technique, which was described in Subsection 1.2.1. Application layer protocol validation is an adequate defense measure.

Chapter 3

Requirements

Before we describe the implementation of our firewall, we will establish functional and non-functional requirements, that determine how our firewall should work. In order to do that, we will need to define a *conversation*. We will use the term throughout the rest of this thesis to refer to a bidirectional communication of two hosts on the network. It is uniquely identified by a *conversation ID* — a combination of source and destination IP addresses, Transport layer protocol number and ports.

3.1 Functional requirements

The main purpose of our firewall is to analyze the payload of Transport layer protocols[1] and ensure the following:

- The transmitted Application layer data conforms to an Application protocol, that is associated with either the source or the destination Transport layer port specified in the conversation.[2] Packets that contain nonconforming data will be dropped.

  The ability to distinguish invalid Application protocol messages allows the firewall to recognize the port spoofing attack or malformed messages, which could also pose a risk.

- Valid Application layer protocol messages will be checked against a list of administrator-defined rules to determine the final verdict, i.e. whether to allow or reject the conversation.

[1] We will focus on TCP and UDP.

[2] E.g. according to IANA assigned port numbers, we expect HTTP messages to be encapsulated in TCP segments with a source or destination port equal to 80.

As we have seen in the previous chapter, Application layer protocols can be abused in many ways and the attacks can be identified by analyzing the protocol messages.

3.2 Non-functional requirements

Besides the functional requirements, we will adhere to a set of constraints, that do not directly relate to the functionality of the firewall.

User-space

OS Linux, which is the base for our firewall, has its virtual memory structured into two parts — the user-space and the kernel-space. The kernel-space is used only by the kernel, whereas the user-space is meant for processes (running application software). The fundamental difference between the kernel and processes is that the kernel can execute privileged instructions, e.g. I/O instructions,[3] load and store operations on control registers, etc.

There are multiple reasons, why implementation in the form of a kernel module was not a feasible option:

Portability User-space programs are easily portable between kernel versions, as they do not interact directly with kernel structures and use the system call interface.

Security User-space programs are isolated from each other and unable to execute privileged instructions, so in case of compromise, the impact is limited. The user-space implementation also allows us to kill the firewall process if necessary, e.g. if it encounters a deadlock or enters an infinite loop. A kernel implementation would require a reboot to resolve this.

Flexibility We aim to implement complex functionality, which would be problematic as a kernel module.

Interaction with networking stack

Since our firewall runs in the user-space and the network stack is implemented in the kernel-space, we had to find a way how to let our firewall examine the incoming packets and issue a verdict. The Netfilter framework provides such functionality in a component called NFQUEUE. It allows a user-space program to receive packets via a Netlink socket and send them back with a verdict [22]. To deliver a packet to the firewall, the kernel has to send it to the Netlink socket and hand over the control to the firewall,

[3] This includes interactions with the network.

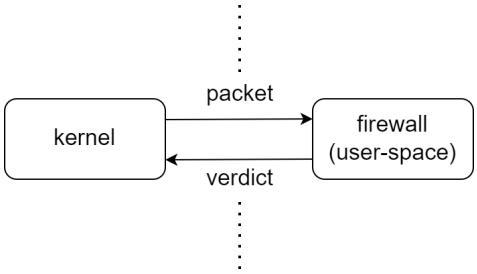

Figure 3.1: Communication between kernel-space and user-space through the Netlink socket

which has to receive it. A similar process is necessary when returning the verdict. The process of handing over control between kernel and firewall, or any other user-space program, is referred to as a *context switch*. It is a time-consuming and resource-intensive procedure and two context switches are needed to process each packet. Our main concern with this approach was the overhead associated with context switching and sending packets between kernel-space and user-space, which could potentially introduce a non-negligible delay. To address this, we wrote a short user-space program, which immediately sends an accept verdict upon receiving a packet through the NFQUEUE. We measured network latency using the ping command with and without the program running and compared the results. The testing environment description, the program and measured values are included in Appendix B.

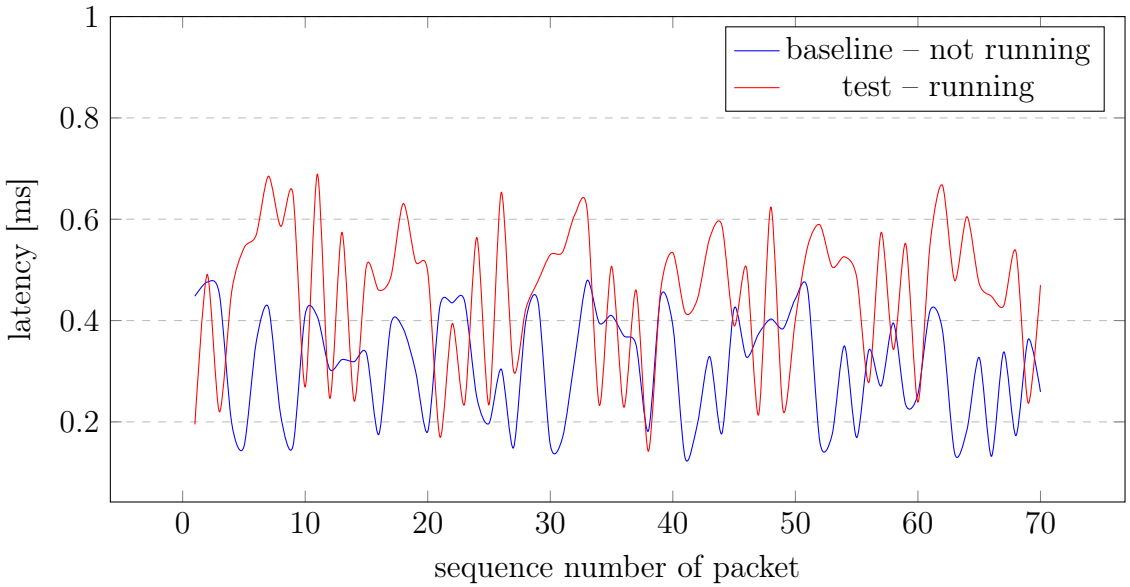

Figure 3.2: Context switch latency test

The average baseline (program not running) latency is 0.310ms and the average test (program running) latency is 0.448ms. The difference is approximately 0.1ms and thus

negligible.

Programming language

We have defined the following requirements for a programming language for the implementation.

Compiled, not interpreted Compiled languages have a performance advantage over interpreted languages — the machine code is generated during compile time, thus no extra overhead is necessary during runtime. Since we want to minimize the traffic processing delay, this is an obvious criterion.

Compilation to machine code Some high-level languages, such as Java, do not compile directly to machine code but use an abstract version of it called bytecode and then compile to machine code during execution. This makes the compiled program independent of the processor architecture, but it has to be interpreted using a virtual machine, which adds runtime overhead.

Library support There should be a library for interacting with the NFQUEUE module. This can greatly simplify interaction with the kernel.

We looked at two candidates who fulfilled these requirements — C and Rust. According to benchmarks, these languages are similar in terms of performance [23]. An important difference between these languages is how they manage the memory. C employs manual memory management, where the programmer is responsible for everything. If done correctly, it usually results in better performance, but mistakes can result in problems such as segmentation faults, memory leaks, etc. One possible approach to automating memory management is to use *garbage collectors*, which check for and free allocated but unused memory[4] during runtime. Rust, however, solves this by using the philosophy of ownership and borrowing, which is enforced by the compiler [24]. The programmer does not have to allocate or free memory manually, it is done automatically when necessary[5] by inserting destructor calls into the program during compilation. This approach removes the runtime overhead associated with garbage collectors and can result in better performance. Ensuring that memory is handled properly is especially important when writing secure software, as a considerable number of severe security bugs in C/C++ applications are a result of memory safety issues [25, 26].

Another safety feature of Rust is its type system, whose design aims to prevent common problems associated with types. For example, the null type is not present.

[4]The memory is no longer referenced.

[5]E.g. when assigning a value to a variable or when the owner of a variable goes out of scope.

Instead, Rust provides the Option type [27], which can explicitly express the absence of data and requires the programmer to check for this condition before they can access the data.

The safety features we described can be bypassed,[6] however, if the programmer abides by them, the compiler can detect and prevent many bugs. Since our firewall will be complex software, and we want to minimize the room for errors, our final choice was Rust.

[6]The borrow checking can be overridden by declaring the code as unsafe or the data within the Option type can be "unwrapped" without the check.

Chapter 4

Design and architecture

This chapter discusses the architecture of the firewall, how the individual modules are structured and issues, which arose during the development. Due to the complexity of the implementation, we decided to structure this chapter as follows. First, we will describe how the requirements from the previous chapter are reflected in the implementation and a high-level overview of the architecture. Then, in Sections 4.1, 4.2 and 4.3 we will discuss the individual phases of packet processing, the involved components of the firewall and design decisions we made during the development. There are certain special cases when the firewall cannot buffer received packets and has to release them without further analysis. We will analyze these cases and solutions in Section 4.4. Finally, we will describe the implementation in detail by following the data flows in the firewall in Section 4.5.

The firewall should operate as a daemon running in the user-space, that continuously receives packets from the kernel, processes them and issues a verdict. Validating an Application layer or L7 protocol can be done by attempting to parse the data according to a formal grammar, that describes the protocol messages. If the parsing fails, the data does not fit the protocol specification and should be dropped. If successful, the parsing also outputs the L7 protocol message content that can be matched against administrator-defined rules that ultimately decide the verdict. The packet processing procedure, i.e. the actions that have to be executed between receiving a packet and issuing a verdict, comprises multiple phases:

1. Preparing input for L7 parsers

2. Parsing the input

3. Applying rules and issuing a verdict

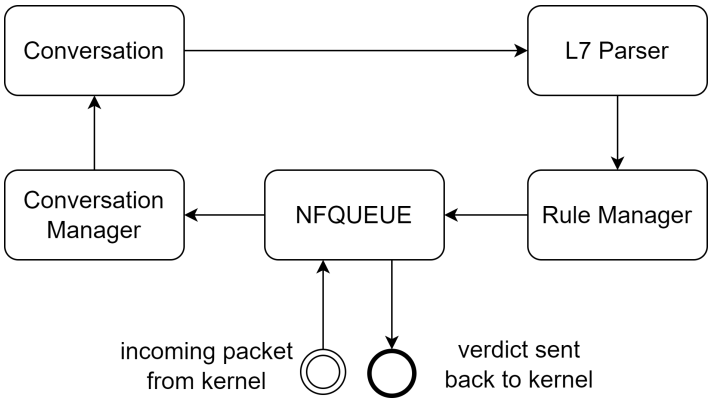

Figure 4.1: Simplified packet processing

The figure shows a simplified set of components, that participate in the packet processing. A packet received from the NFQUEUE is first preprocessed by the Conversation manager and assigned to a Conversation. Data from each Conversation is processed by an L7 parser and if successful, the parser output is sent to a Rule Manager module, that performs the rule matching and issues a verdict. We will now discuss the individual phases and problems, which we encountered.

4.1 Preparing input for L7 parsers

Before we can run the L7 parsers, the firewall has to preprocess the incoming packets. In other words, it has to:

1. Parse the L3 and L4 headers.
2. Assign the packet to the correct conversation — this can be done using the information obtained in the previous step.
3. Provide the L4 payload to the parser as input — the payload is identified in the first step.

Several problems were encountered when designing the implementation of this process.

4.1.1 TCP specifics

Since TCP works as a byte stream, the segment boundaries have no relation to the boundaries of the encapsulated Application layer protocol messages. In order to correctly identify the L7 protocol data unit (PDU) boundaries, we cannot evaluate the segments individually, we have to reconstruct the entire byte stream. Only then can we correctly interpret the Application layer data.

Before we describe the reconstruction, we first need to mention some aspects of TCP operation. TCP is a connection-oriented protocol, so before any data is transmitted, the connection is established by a *three-way handshake*. Each SYN packet contains the

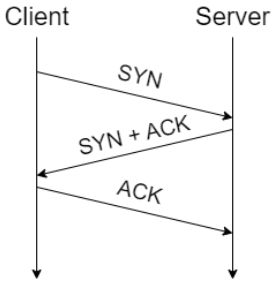

Figure 4.2: Three-way handshake

first sequence number for the given direction, i.e. client-to-server and vice versa. A *sequence number* is assigned to each byte in the stream and is used by the receiving host to reassemble the TCP segments[1] in the correct order. The connection is either closed gracefully in a process similar to the handshake but with FIN instead of SYN flags, aborted using the RST flag or the connection times out.

The module responsible for TCP conversation reconstruction will have the following two responsibilities:

Tracking the state of the connection Based on the flags in the incoming packets update a state machine.

Processing segment payloads Pair the segment payload with the sequence number of the first byte, that was extracted from the segment header and add it to the reassembled stream.

Figure 4.3 shows the TCP conversation states and what flags trigger the transitions between them, with two exceptions that were omitted for clarity:

- RST flag — The RST flag is used to abort the connection in any given state. If we receive a TCP segment with the RST flag set, we immediately transition to the Finished state.

- Anomalous state — All other flag combinations mean that the conversation is anomalous and trigger a transition to a designated Anomalous state.

[1]The segment represents a series of consecutive bytes, so the only information necessary for reconstruction is the sequence number of the first byte and the length of the payload.

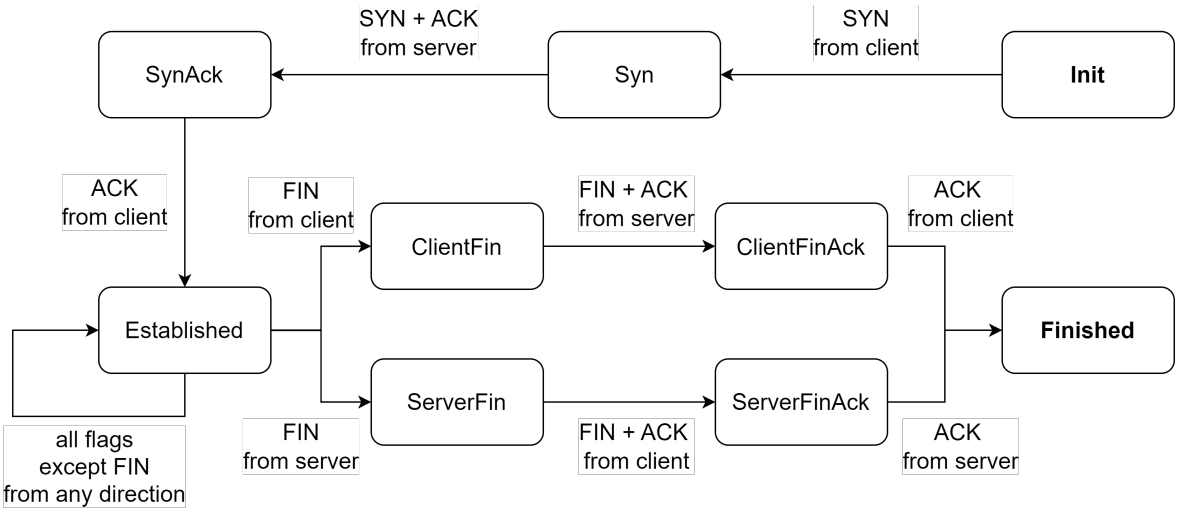

Figure 4.3: Valid TCP state transitions

4.1.2 Incomplete input

As we have discussed, TCP segment boundaries have no connection with the Application layer protocol message boundaries. This means that we have to run the L7 protocol parser on an entire byte stream. We do not know that the stream is complete until the connection closes. There are two options how to address this:

1. Buffer the segments and wait until the connection closes (or times out), then run the parser on complete input (or what we managed to reconstruct)

2. Adapt the parser to the streaming nature of TCP, so we can run it on an incomplete stream.

The first option seems simpler, but is not feasible, as the communication between the hosts is bidirectional. TCP and many Application layer protocols depend on acknowledgements or other feedback from the other host, e.g. the TCP segments have to be acknowledged by the receiver, or the HTTP server sends data as a response to a request. If we tried to buffer the entire communication, the feedback would be absent and the communication would come to a halt. On the other hand, the streaming parser solution is more complex in terms of implementation but allows us to examine the incoming traffic as soon as it arrives. This is desirable, as it keeps the memory requirements low and ensures early detection of attacks such as garbage flood.[2] This solution, however, is not completely free from problems caused by buffering TCP segments. The problematic cases are further analyzed in Section 4.4.

The most challenging aspect of the chosen approach was the interface between an L7 parser and a Conversation. We explored two possible solutions.

[2]The attacker attempts to overwhelm the victim with random data, i.e. garbage.

Trial and error

With this approach, the Conversation tries to call the parser every time it receives a new segment.[3] The parser will return one of three possible results:

success the input represented a complete L7 PDU that matched the protocol

fail the input did not match the protocol specification

incomplete the input appears to match the protocol but does not represent a complete message

If the result is *fail*, the firewall drops the conversation. In the case of *success*, the parsed message is further evaluated based on rules. The *incomplete* result instructs the Conversation to try again with more input. In other words, the parser is always given a larger prefix of the stream until it returns a definitive answer regarding an Application layer protocol message. A very similar approach is implemented by L7 parsers in Suricata. The main difference is that instead of the incomplete result, the parser returns the number of bytes that are required to obtain the definitive answer [28].

Threads

In this approach, we leverage that a thread can be blocked and when it resumes execution, its state will be restored. So in case we run out of input, we can interrupt the parser and resume exactly where we stopped when more data is available. We can implement this behavior with the producer-consumer pattern, where the Conversation sends the incoming input via a producer and the parser receives it from a consumer.

With this approach, the number of threads is approximately equal to twice the number of active conversations, as we need to run a parser for each direction of the conversation. In case the firewall host has sufficient resources, the parsing can be parallelized. On the other hand, this approach adds the need for more context switching on the thread level, which could introduce computational and time overhead, that may become significant when processing numerous simultaneous conversations. To address this concern, we wrote a short program, that implements the producer-consumer pattern with many consumer threads. We tested whether the operating system can handle a large number of threads and measured how it affects CPU usage. The program and the testing methodology description are included in Appendix C.

The results have shown that the operating system can handle 10000 consumer threads, which should be sufficient for most intended deployments. The effect on CPU

[3] Except for retransmissions and cases, where a segment is missing, i.e. we do not have a continuous stream.

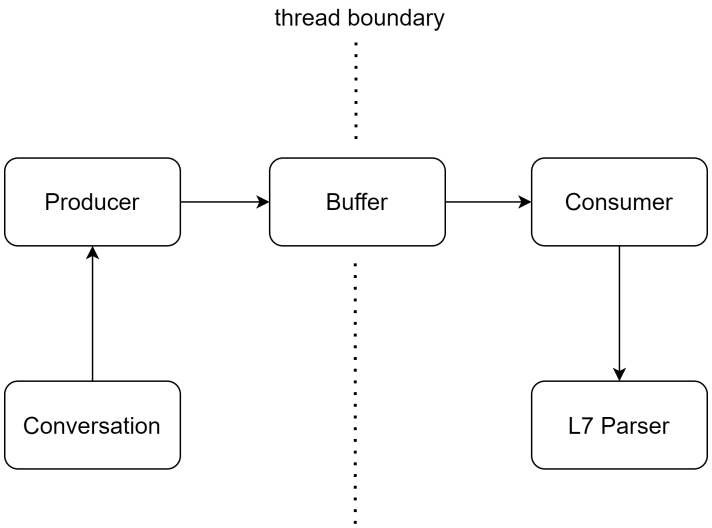

Figure 4.4: Producer-Consumer pattern

usage in a real deployment is difficult to estimate as it largely depends on how the firewall is deployed. This is further discussed in Appendix C.

Now that we outlined the architecture of the firewall, we are going to describe the design of individual modules and interfaces in more detail.

4.2 L7 parser

4.2.1 Parser design

There are two main approaches when it comes to writing a parser. It can be either generated from a formal grammar specification or handwritten. We explored both options.

Generated parser

Tools such as bison or yacc are often used for generating parsers mainly for programming languages, but they can be used for other purposes as well. However, the network communication processing and the programming language compilation are incompatible processes. The parsing of programming languages begins with lexical analysis, which separates the input into tokens, which are further processed. This is unnecessary and often not possible with network protocols, especially those encapsulated in TCP, as the input is incomplete until the connection closes. For this reason, we decided against generated parsers.

Handwritten parser

In general, handwritten parsers tend to be more error-prone as the entire implementation is done by the programmer. However, this is not an issue with network protocols, which are by design straightforward to analyze, as they have to be parsed in real time. To illustrate this, one of the most complex aspects of parsing DNS messages is the reconstruction of compressed domain names [29]. Without the compression, the procedure for parsing the name is simple. The name is represented as a sequence of labels, each preceded by a length byte that holds the length of the label. The sequence is terminated by a length byte with a zero value. This format can be parsed using a simple while loop. When using the compression, the names can now contain pointers to other names within the DNS message. The modified parsing procedure thus involves a jump based on the pointer, which can be done with recursion. The final implementation is a straightforward recursive method with a while loop. Together with the use of threads for handling incomplete input, the simplicity of the Application layer protocols allows us to implement the L7 parsers as intuitive procedural code.

4.2.2 Parser interfaces

Another important aspect of the parser design is the interface with the consumer that provides input and the format of parsed data, which is further processed by a module that manages firewall rules.

Consumer interface

Besides forwarding the input to the parser, the producer-consumer pattern is responsible for encapsulating the specifics of the Transport layer protocol. For this reason, we implemented two versions of the pattern. In the case of UDP, the datagram boundaries represent a logical delimiter and the parser requests one datagram at a time using a method named get_next_dgram() implemented by the consumer. In the case of TCP, the Application layer data is a stream without delimiters reassembled from individually transported segments. The reconstruction mechanism implemented by the producer-consumer pattern is discussed in detail in Section 4.5.2. The TCP-specific consumer provides the parser with two methods:

get_bytes(n) The consumer will return the next n unread bytes of the stream.

get_until(pattern) The consumer will return unread bytes that precede the first occurrence of the pattern in the stream. This method can be used when parsing HTTP header fields, which are delimited by the new line and carriage return control characters.

This way, the stream reconstruction remains encapsulated, and the parser code is cleaner and more intuitive.

Output format

The firewall needs to be able to analyze and apply rules to different Application layer protocols. Each protocol has a different message format, so we had to find a suitable abstraction for the structures. We decided to represent the parsed data using a key-value map, where the key is a string, e.g. "src_ip", "content-length", etc., and the value can be one of multiple types. We defined a set of possible value types, which cover all the necessary structures found in Application layer protocol messages. This set contains primitive types, such as string and integer, but also aggregate types, such as list and map. The list type can contain any value type, so even a recursive list is possible.[4] The map type is similar to the list in that it can contain any of the values with the key being a string.

Mandatory and optional keys and associated value types are defined for each of the analyzed Application layer protocols and listed in Appendix D.

4.3 Rules

We will now describe how the firewall defines rules, what components comprise the implementation and how they are applied.

4.3.1 Rule design

Our firewall allows the administrator to define the rules in the form of a configuration file and reload them by sending a SIGHUP signal to the daemon. The file can contain two types of statements — declarations and rules.

A *rule* consists of a verdict (allow, drop or allow pdu[5]) and a condition. The condition resembles a propositional formula, i.e. it is built from atomic conditions using logical conjunction, disjunction and negation. The atomic conditions offer a set of operations, that can be performed with the values returned by the parser. For example, an atomic condition could check if a value determined by a specified key is of the string type and equals another string, matches a regex, etc.

Unlike a rule, a *declaration* does not directly affect the evaluation of a parsed PDU. Its purpose is to configure one of several modules, which provide stateful filtering

[4] An example of a valid list is [true, "abc", 45, false, [1.1.1.1, 32]].

[5] This verdict releases the currently processed PDU to advance the conversation and thus obtain the next PDU.

functionality or simplify the configuration. The following modules are available in our firewall:

Rate limit This module is similar to the hashlimit module offered by iptables. The administrator can configure how many Application layer PDUs that match specified conditions are allowed during a specified time interval. For example, we could configure, that the firewall will allow 10 DNS responses containing MX records with specific destination IP address per minute. This functionality can be used as a form of DDoS protection.

Aggregation The purpose of this module is to count PDUs that match defined conditions. Using these values, rules can define arithmetic expressions and compare results. For example, this could be used to detect DNS tunneling as discussed in Section 2.1.1. The Aggregation module could count the number of all DNS requests and the number of DNS requests for MX records from a specific host. These two values can then be used in a rule, which computes the ratio of MX record requests and checks if it is within a threshold.

IP address list This module allows the administrator to create named lists of IP addresses and then use them to define conditions that match when an IP address is found in a list.

A more detailed description of semantics and formal specification of the configuration file syntax can be found in Appendix E.

4.3.2 PDU evaluation

A component named *Rule manager* receives parsed PDUs from the L7 parsers. It is responsible for determining the verdict for a conversation based on the configured rules and the data from the aforementioned modules. When a new parsed PDU arrives, the first step is to check if there are any matching conditions in the Rate limit module. In case the PDU matches and the limit for a given condition has been exceeded, the conversation will be dropped. Otherwise, the PDU will be checked against conditions defined for the Aggregation module and any matching counters will be updated. The last step is to compare the PDU with the configured rules. The Rule manager stores and evaluates the rules in the same order as they were specified in the configuration file. If the PDU does not match any rule a default verdict will be returned. Otherwise, the verdict from the first matching rule will be returned.

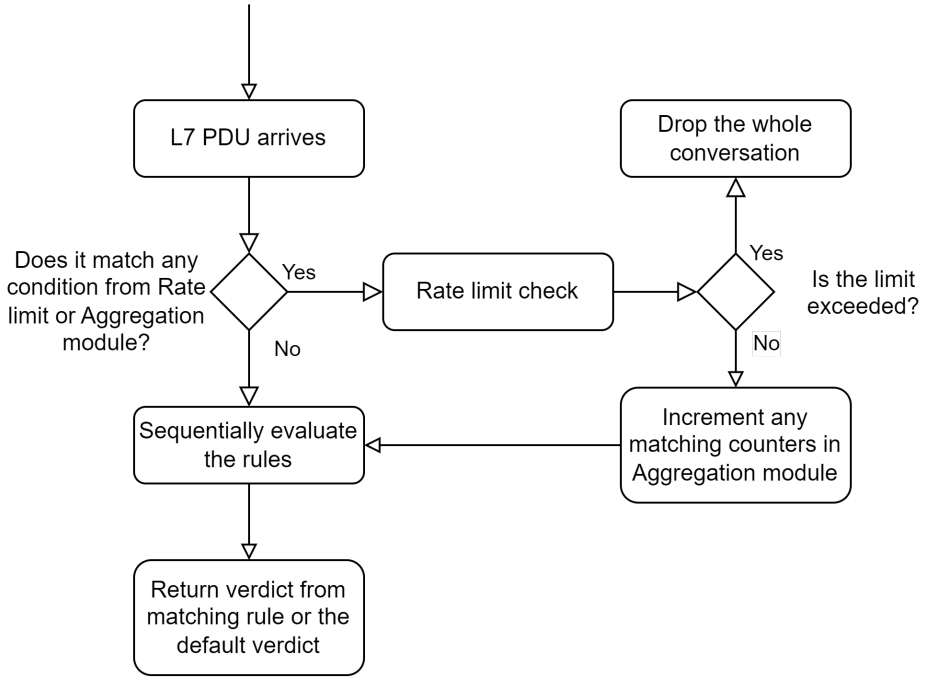

Figure 4.5: Rule manager decision process

4.4 Allowing packets before final verdict

There are certain situations, where letting some packets through the firewall is necessary. We will now describe them, and explain the problems that occur if we do not allow the packets to pass through the firewall and how we address these situations.

4.4.1 TCP handshake

As we mentioned before, TCP is a connection-oriented protocol, and it has to perform the three-way handshake before any Application layer data can be transmitted. The SYN + ACK packet sent by the server is a response to the first SYN packet sent by the client. If our firewall captures and buffers the first SYN packet, the server does not receive it. The server does not know about any connection attempt, so it does not send any SYN + ACK reply and the client cannot finish the handshake. This leaves the communication effectively in a deadlock.

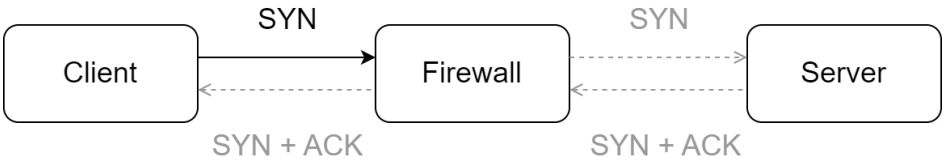

Figure 4.6: Three-way handshake deadlock — the greyed-out messages are never sent

This problem can be resolved by letting the handshake packets pass through the firewall while only tracking the state of the TCP connection, i.e. validating the order of packets and the flags. After the handshake successfully finishes, both the client and the server can start sending data which will be buffered and parsed by the firewall.

4.4.2 TCP sliding window exhaustion

We identified another situation, where a deadlock may occur. TCP utilizes the concept of *sliding window*. The size of the sliding window specifies how many unacknowledged bytes can be sent to a host. The receiver regulates the window by sending the window size and acknowledgement number in the TCP header. This way, it can shift or "slide" the window by incrementing the acknowledgement number and changing the size of the window, even to zero if it cannot process any more data.

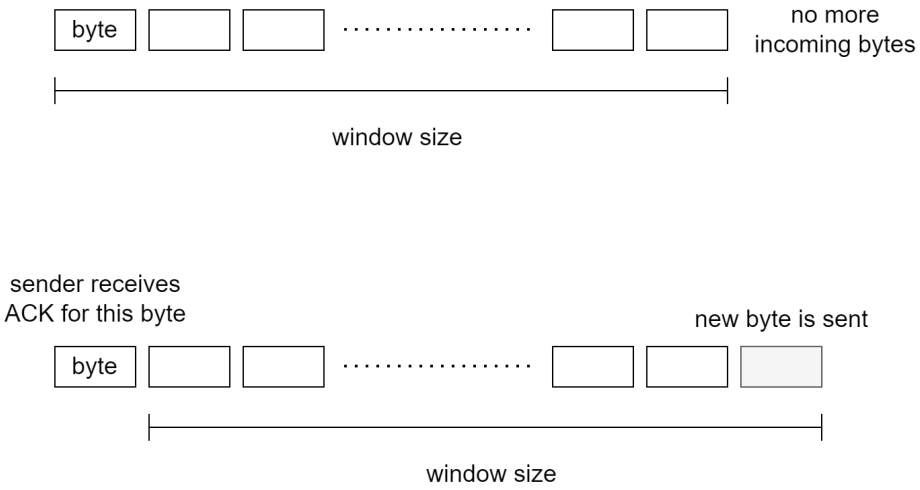

Figure 4.7: Sliding window mechanism

To prevent unwanted traffic from reaching its destination, our firewall buffers received packets until it reaches a verdict. We will examine a situation, where the conversation could come to a halt due to the TCP window becoming full.

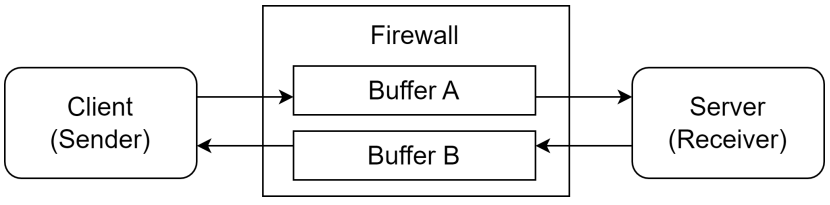

Figure 4.8: Firewall buffers

Consider an HTTP client sending a request that is larger than the TCP window determined by the server. Our firewall buffers the incoming packets from the client in

Buffer A and continually parses them. The server does not send any acknowledgements as none of the buffered packets from the client arrive. When the client exhausts the window, it stops sending new packets. If our firewall does not have enough data in Buffer A to reconstruct an entire HTTP message, it will halt waiting for more input, which never comes.

This potential problem was discovered during the implementation phase, so we had to design the solution with respect to the existing architecture. Each direction of the conversation is processed independently, i.e. there are two producer-consumer pairs that move the input between threads and two parsers. Each parser-consumer pair is isolated in its own thread. In order to prevent the firewall from halting, it has to be able to recognize the window exhaustion and respond.

The exhaustion can be easily detected by tracking the acknowledgement numbers and window size values that are included in the TCP headers sent by the receiver and checking if sequence numbers from the sender do not exceed the window. To keep the conversation from stopping, the sender has to receive an acknowledgement from the receiver, so more bytes can be sent. Two situations can occur. In case the Buffer B is not empty, the firewall should release the first buffered packet. If it is empty, the firewall first has to obtain an acknowledgement from the receiver. This can be achieved by releasing the first packet from Buffer A and when the acknowledgement arrives in Buffer B, releasing it to the sender. In both cases, an interaction between the threads is necessary.

Since the input for each direction is processed in a separate thread, we had to decide which thread should be responsible for releasing the packets.

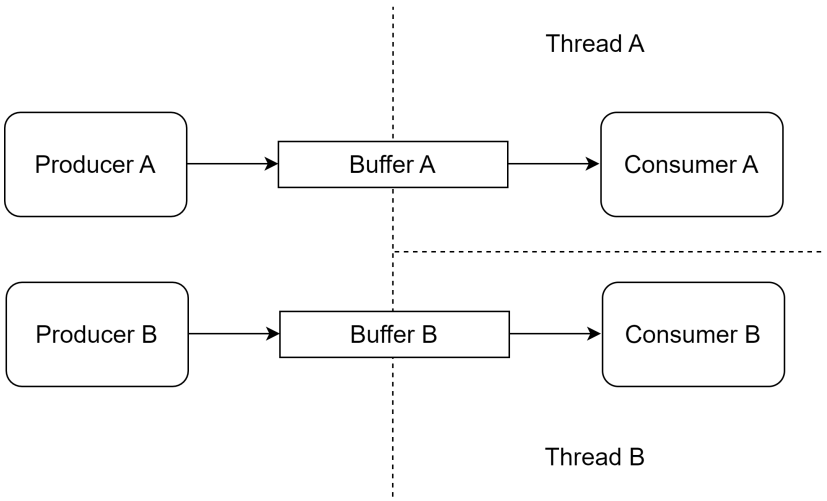

Figure 4.9: Producer-Consumer pattern in detail

In the following analysis, we assume that Thread A is the one that detected window exhaustion. One possible solution was to let it access Consumer B or Buffer B directly

and release the first packet. We rejected the option with direct access to Buffer B, as it would interfere with the operation of Consumer B. The Thread A could potentially release a packet from Buffer B, that has not been processed by the parser. Thus, the parsing process would be unreliable as the input stream may be missing sections.

The other possible solution, where Thread A controls the packet release would involve accessing Consumer B, i.e. Consumer A would have a reference to Consumer B and could call a method to release a packet. The problem from the previous approach is irrelevant here, as the Consumer knows which packets were already parsed. However, this approach would require a circular reference between the Consumers, which could introduce further deadlocks.

After further analysis, we decided in favor of a solution, where the two threads would share only a structure with the following flags, which govern the releasing of packets:

Release possible This flag signals to the other thread that there is at least one buffered packet, that has been parsed and thus can be released.

Release request This flag is set by a thread to request packet release from the other thread.

The structure contains these flags for each direction of the conversation. The threads would also have a reference to a conditional variable in the other thread.[6] They remain otherwise separated, with each having access only to its consumer and parser structures.

The complexity of this approach lies in determining when the threads should check the state of the flags, especially the one signalizing a release request. To properly analyze this, we identified what states the Consumer thread can be in and how it transitions between them.

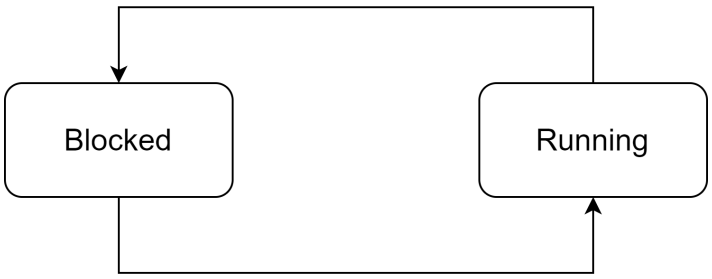

Figure 4.10: Consumer thread states

For simplicity, we omitted the initial and final, or terminated, states. The thread can be in the following states:

[6]This is necessary, as the other thread may not be running when we need to request a packet release.

Running The buffer contains input, and it is either being prepared by the Consumer[7] or processed by the parser.

Blocked The buffer is empty, so the thread is not executing.

The thread can transition to the Blocked state if it tries to get data from the buffer, but it is empty. It resumes the Running state as soon as the Producer fills the buffer and notifies the Consumer. If a window exhaustion occurs in Thread A, both Consumer threads will eventually reach the Blocked state and without any modification, this would mean that a deadlock occurred. To prevent this, Consumer B checks the state of the release request flag before entering the Blocked state. This avoids the deadlock, if the Thread B was in a Running state when the release request was issued. This is not always the case. In the example situation with a large HTTP request, that was discussed earlier, the Thread responsible for parsing the response does not have any data available. It enters the Blocked state immediately after being initialized. For this reason, the Consumer in the request parsing thread must be able to notify and "wake up" the request parsing thread. It will enter the Running state, but since there is still no data ready, it will only check the release request flag and transition back to the Blocked state.

4.4.3 TCP retransmissions

Another aspect of TCP, which we had to address, are retransmissions. If a sender does not receive an acknowledgement in a retransmission timeout, it assumes the packet did not reach the receiver and sends it again. We had to decide how the firewall should handle this situation. A retransmitted segment can be identified by its sequence number, which is lower than the next expected sequence number. When we receive such a segment, two cases can occur — the retransmitted segment was either already processed and released from the firewall or it is still in the buffer. Before we analyze these cases we will establish an assumption, that the L7 parsers have *sufficient granularity*. We will illustrate this on an abstract Application layer protocol, which uses TCP. Let's assume, that host A sends some data to host B and then waits for a response from host B before sending any further data. An example of such a protocol is SMTP. Our firewall buffers the data from host A and tries to parse it. If the L7 parser cannot parse and return an L7 PDU from the available data and waits for more input, the segments sent by host A never reach host B and the necessary reply from host B is never sent. The conversation comes to a halt. If the parser is designed with sufficient granularity, the parser can always return an L7 PDU before the sender (host A) needs a response from the receiver (host B) and the halt is avoided.

[7] E.g. the Consumer is reassembling TCP segments.

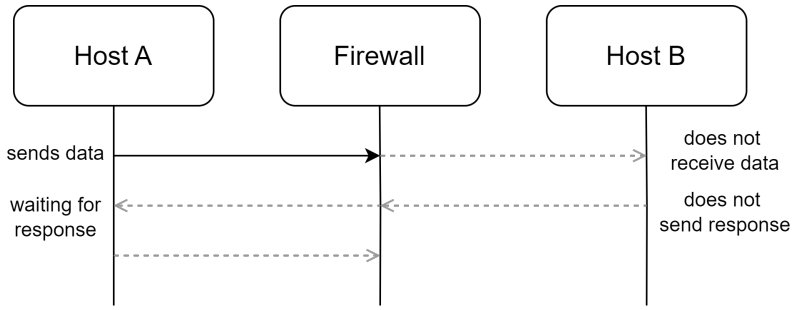

Figure 4.11: Insufficient parser granularity

Now we can analyze the two cases, that can occur when we receive a retransmitted segment:

- The retransmitted segment was already released from the firewall buffer. This means, that we already parsed and released this segment earlier, so it either did not reach the receiver, or the buffer for the other direction is withholding the acknowledgement. If we assume the sufficient granularity of the L7 parsers, the second situation cannot happen. The correct behavior in this case is to release the segment without further analysis.

- The retransmitted segment is still in our buffer. This could happen in multiple situations:

  - **The parser is waiting for more input.** This cannot happen under the assumption of sufficient granularity.

  - **The firewall is processing a large L7 PDU.** In this case, the first segments could theoretically time out, before we can reach a verdict and release the segments.[8] In this case, we should quietly drop the retransmitted segment as the firewall has it buffered and will eventually release it.

  - **The data does not correspond with the protocol definition.** For example, HTTP does not limit field length but rather uses delimiters. If the received data does not contain the delimiter, e.g. there is a port spoofing attempt, the parser will wait for more input. The sender will eventually try to retransmit the segments, but our firewall quietly drops them, as described above. The sender will eventually close the connection and the record of the conversation in our firewall will time out.

[8] If the verdict allows it.

HTTP

To satisfy the sufficient granularity assumption for HTTP, we had to use a modified parsing logic. The HTTP messages comprise headers and body. The body, especially in case of an HTTP response, may be arbitrarily long and buffering it entirely would not be possible. For this reason, our HTTP parser returns a parsed PDU after successfully reading the HTTP message headers. Then it enables *auto drain* mode, where it parses the message body, but every segment is automatically allowed. As soon as the parser finds the end of the message body, it disables the auto drain and returns the body as the next parsed PDU, which is then evaluated by the Rule manager.

4.5 Packet processing

We will now describe the data flow in the firewall, i.e. what happens with a packet from the moment it is received from the kernel to the moment when the firewall sends back a verdict. As we established earlier, a conversation is uniquely identified by IP addresses, Transport layer protocol type and ports, i.e. the conversation ID.

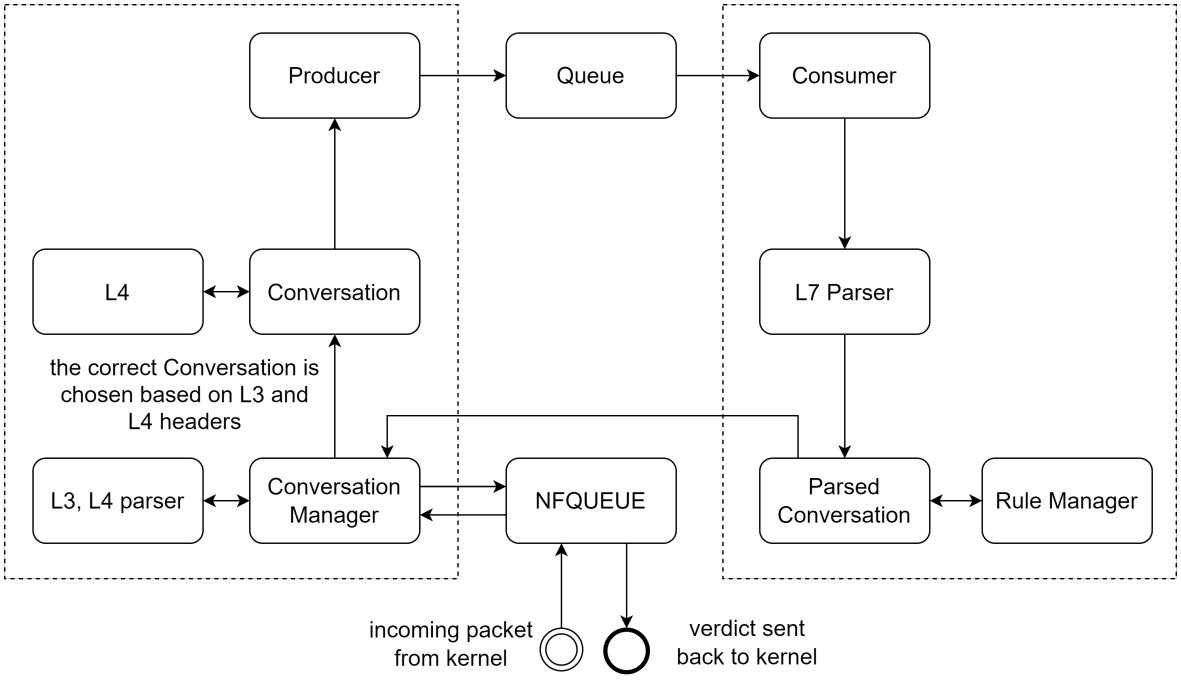

Figure 4.12: Packet processing overview

Figure 4.12 shows how a packet can travel through the individual components of the firewall. The dashed lines represent thread boundaries. Structures and other components, that do not interact with the packets directly, were omitted for clarity. For instance, a conversation is bidirectional, and data from each direction is processed

by a separate thread. However, these threads are identical in terms of components and data flow, so we focus only on one of them.

4.5.1 Preparing input for L7 parsers

The thread on the left in Figure 4.12 runs in a loop. It receives packets from the kernel and either sends them to the thread with an L7 parser (the one on the right) or back to the kernel in some cases that will be discussed.

The received packets are first processed by a module named *Conversation manager*. Its purpose is to maintain records of all active conversations and assign the incoming packets to correct conversations for further evaluation. The records are kept in a hash table, where the keys are conversation IDs and the values are *Conversation* structures, that represent and manage active conversations. The records in Conversation manager are removed when a conversation times out[9] or finishes, e.g. TCP gracefully closes the connection. The procedure done by the Conversation manager comprises the following steps:

1. The packet is first sent to the *L3,L4 Parser*, which attempts to extract information from the Network and Transport layer headers. If successful, the conversation ID can be constructed and the firewall moves to the next step. Otherwise, a default verdict for this case will be applied, and the packet will be sent back to the kernel.

2. If there is a record with a given conversation ID in the Conversation manager hash table, the packet will be sent to the corresponding Conversation structure for further evaluation. If there is no matching record, it will be created.

3. The Conversation structure can return the packet together with a verdict in case the conversation is definitively allowed or dropped.[10] In this case, the Conversation manager will finish the processing by sending the verdict back to the kernel. The next step depends on whether the conversation is tracked or untracked.

Some conversations will be *untracked*, which means that they are not processed by the L7 parsers. This could be because they use an Application layer protocol, that is not recognized by our firewall or the administrator does not wish to analyze them for some other reason. Tracked and untracked conversations are identified based on Transport layer ports. A configuration file contains port and Application layer protocol mappings. If a conversation uses a port that appears in these mappings, the Application layer data will be validated against the specified protocol. All other conversations will be

[9] Each record has a time-to-live value, which is reset when a packet is received. If it reaches zero, the record is automatically removed and the conversation dropped.

[10] A decision based on the rules configured by the administrator or Application layer protocol validity, e.g. if the data does not match the protocol, the conversation is dropped.

untracked. The untracked conversations have a record in the Conversation manager table, but the Conversation structure always returns a default verdict for an untracked conversation. We will now continue to examine the data flow for a tracked conversation. As the next step, the packet is processed by the Conversation structure:

1. If the conversation does not have a definitive verdict assigned, the packet will be processed by *L4* module. This module tracks information associated with the Transport layer protocol based on data from parsed Transport layer headers. For example, in the case of TCP, this module is responsible for tracking the state of the connection based on the flags in TCP segment headers. This allows the firewall to start buffering only after the three-way handshake successfully completed and terminate the parser threads if the hosts decide to close the connection. In the case of TCP, the L4 module also prepares the sequence numbers, which are used to reconstruct the byte stream.

2. If the packet contains Application layer data, e.g. it is not part of the TCP handshake, and the conversation uses a port, that is tracked by our firewall, the packet payload and an ID will be sent to the correct parser thread via the producer-consumer pattern. The original packet stays in the Conversation structure in a hash map where the ID serves as a key.

4.5.2 Producer-consumer

In the case of TCP, the producer-consumer pattern, specifically the consumer component, encapsulates the complexity of the stream reconstruction. The *Producer* accepts segments together with their sequence numbers and sends them through the *Queue* to the *Consumer* in the parser thread. The Consumer stores the packets in a priority queue, where the sequence number serves as the priority value. The Consumer also stores a *current sequence number*, which belongs to the next byte that should be processed. When a byte is processed, i.e. sent to the parser, the current sequence number is incremented. This way the Consumer can detect missing segments and correctly handle retransmissions as described in Section 4.4.3.

The possibility of incomplete input is elegantly handled by the producer-consumer pattern. If the Queue is empty, the Consumer blocks until the Producer prepares new data. This way the parser can resume execution in the same state in which it ran out of input.

In the case of UDP, the design is considerably simpler, as the producer-consumer forwards the datagrams to the parser in the same order as they arrived.

4.5.3 L7 parser result processing

The L7 parser runs in an infinite loop processing input from the Consumer. When it returns a result, a *Parsed conversation* structure stores it and executes the following procedure:

1. The result could be an error, if the Application layer data did not match the protocol specification. In this case, the conversation is immediately terminated and dropped.

2. The result could also indicate the end of input, which means that the conversation was terminated by the underlying Transport layer protocol, e.g. TCP closed the connection.

3. The last option the parser can return is a successfully parsed L7 PDU which is sent to the Rule manager for further evaluation. This process is described in more detail in Section 4.3.2.

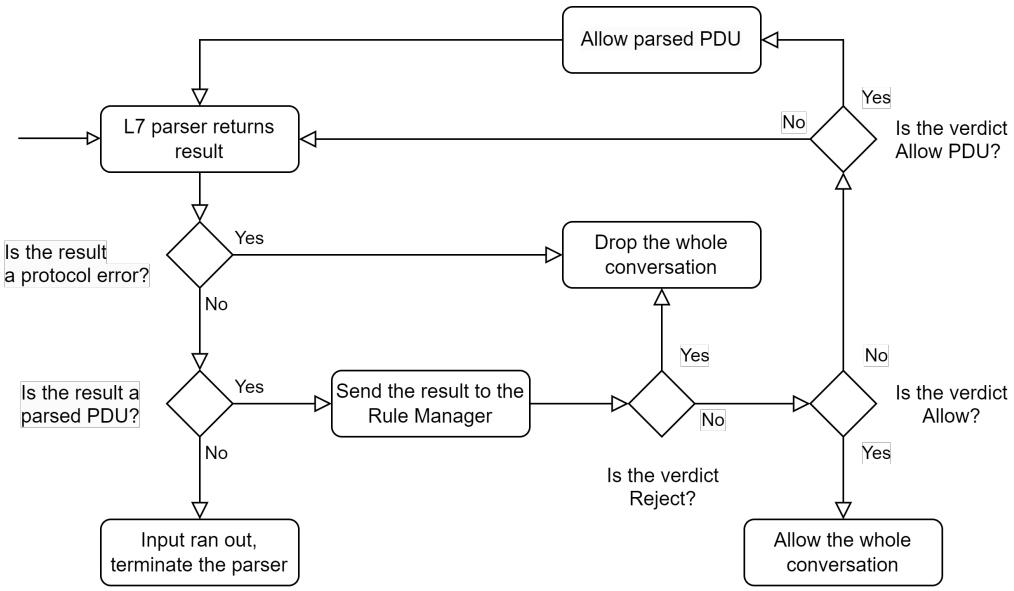

Figure 4.13: Parsed PDU processing

The verdict issued by the Rule manager determines what happens with the buffered packets:

Allow All buffered packets are allowed through the firewall. Any new packets that belong to this conversation are also allowed.

Drop Same as Allow, except the buffered and future packets are dropped.

Allow PDU The buffered packets are allowed, but all new packets are buffered and parsed.

In case of a final verdict (Allow or Drop), the parser thread terminates and the Conversation manager is responsible for issuing verdicts to new packets.

4.5.4 Issuing verdict from parser thread

As we described in previous subsections, the firewall runs a loop in one of the threads, where it tries to receive a packet from the queue and then process it. The parser threads are responsible for returning verdicts for packets, whose payload has been processed, so they need a way to access the queue structure and issue the verdicts. We implemented this mechanism using channels,[11] which allow multiple producers (in this case parser threads) to send messages to a single consumer (thread with loop). The parser threads send the packet ID[12] and a verdict via a channel to the thread with the queue structure running the loop. In each iteration of the loop, this thread checks the channel for any pending verdicts and applies them before receiving the next packet from the queue. The process can be expressed as the following pseudocode:

```
while !terminate {
  if !pending_verdicts.is_empty() {
    issue_verdicts()
  }
  packet = recv()
  process(packet)
}
```

NFQUEUE library issue

During testing, we discovered a problem with the NFQUEUE library. Our firewall uses a structure, which represents the queue for communication with kernel. Besides others, this structure offers these two methods:

- recv() – blocking call, which waits until there is a packet in the queue and then returns it
- verdict() – non-blocking call, returns a packet to the kernel with a verdict

The problem becomes apparent when the firewall receives too few packets. In this situation, the receiving thread blocks on the recv() call and does not issue verdicts, until a new packet is received. Only then the receiving thread enters a new iteration of the loop and issues the pending verdicts. This causes an increase in latency and in certain cases, the communication can time out.

[11] `https://doc.rust-lang.org/std/sync/mpsc/`

[12] Where the packet ID comes from is described at the end of Subsection 4.5.1.

We implemented a solution, which allows the parser thread to "request" the recv() method to return. Internally, the recv() method uses the standard Unix recv() system call. We decided to leverage its behavior when a signal is received. We chose an unused signal and registered a sighandler. We did not set the SA_RESTART flag in the sigaction() call. This means, that if the signal handler executes when the recv() system call blocks, the recv() will return with an error and the execution of the program continues. To ensure that the signal is handled by the blocked thread, we mask the signal in all other threads, i.e. the parser threads. If a parser thread wants to issue a verdict, it simply sends the signal to the firewall process, which interrupts the recv() and ensures that the verdict is applied as soon as possible.

Chapter 5

Testing

In this chapter, we will demonstrate how our firewall processes legitimate traffic and mitigates some of the Application layer protocol attacks described in Chapter 2.

5.1 Methodology

The testing environment was set up on a physical host running Linux Ubuntu server. We created two routed virtual networks and two virtual machines running Linux Fedora, each in one of the networks. On one machine, we installed a web server (Nginx) with a sample web page and a DNS server (bind9), which was the authoritative server for zone example.com. The firewall was installed on the physical host with manually configured iptables rules in the forward chain:

```
iptables -A FORWARD -i virbr2 -j NFQUEUE --queue-num 0 --queue-bypass
iptables -A FORWARD -o virbr2 -j NFQUEUE --queue-num 0 --queue-bypass
```

During each test, we captured all network traffic that was sent and received on the virtual interfaces using tcpdump. These captures were used to evaluate the test results and are included in Appendix F.

5.2 Functional tests

The following tests demonstrate and verify the functionality of the firewall's Application layer validation and further inspection capabilities.

5.2.1 Legitimate traffic

First, we tested the firewall with legitimate HTTP and DNS traffic. The web server was configured with a simple web page and the DNS server was an authoritative server for zone example.com. The web page files and the DNS zone configuration file can be

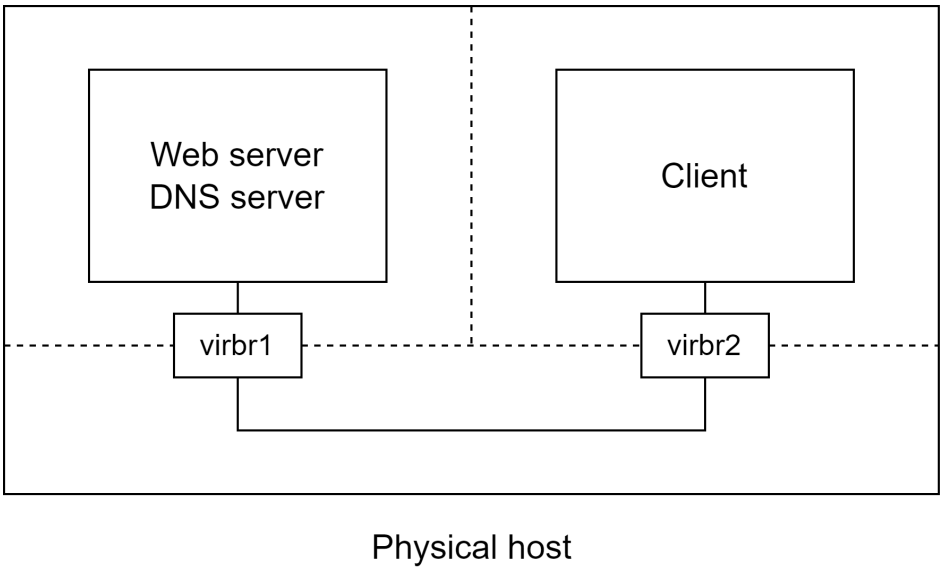

Figure 5.1: Testing environment

found in Appendix F. The firewall ruleset was left empty and the filtered ports were set to default values, so any messages with port 80/TCP matching HTTP protocol or with port 53/UDP matching DNS would be allowed through the firewall. All other firewall settings were left with default values. We tested the Application layer protocol validation with the following procedures:

HTTP The client virtual machine requested index.html file using command `wget http://192.168.124.19`

DNS The client requested DNS records for www.example.com using command `dig @192.168.124.19 www.example.com`

The packet captures and firewall logs in Appendix F show that the firewall correctly identified the protocols and the communication was successful.

5.2.2 Port spoofing

Next, we verified whether the Application layer protocol validation can detect port spoofing. The web and DNS server and the firewall had the same configuration as in the previous test. The client sent the following:

HTTP The client virtual machine attempted to connect to the web server with SSH using command `ssh 192.168.124.19 -p 80`

DNS The client sent a UDP datagram to the DNS server, which did not match the DNS query format using the command `echo -n "not a dns message" > /dev/udp/192.168.124.19/53`

The packet captures and firewall logs in Appendix F show that the firewall dropped the incoming traffic due to a protocol error and the Application layer data never reached the servers.

5.2.3 DNS DDoS

As we discussed in Section 2.1.2, DNS can be used to perform reflection and amplification attack. To illustrate how an L7 firewall can mitigate an Application layer DDoS attacks like this, we implemented the Rate limit module described in Sections 4.3.1 and E.0.5. In this test, we demonstrate its functionality in a simplified example. We configured the firewall to limit the amount of DNS responses with MX records to 1 per minute.

```
let limit_MX : RATE 1/60s records LIST has_item MAP has_id type NUM = 15;
```

This is a declaration named limit_MX, which limits the number of L7 PDUs per minute, which satisfy the following condition:

- the PDU has a value of type list under the key “records”
- the “records” list contains a value of type map
- the map contains a value under the key “type”
- the “type” value is equal to 15

In other words, this condition matches PDUs, which are DNS replies and contain an MX record. During the test, the client sent five DNS queries requesting the following records:

- twice MX records for mail.example.com
- once A records for www.example.com
- twice MX records for mail.example.com

The first three queries were sent shortly one after another, the last two were sent at least one minute after the first MX record query. The logs and packet captures show, that responses to the second and last MX record query failed due to the configured rate limiting. The other MX record queries were successfully checked by the Rate limit module and the A record query was only validated by the DNS message parser as expected.

5.2.4 HTTP policy enforcement

In this test, we demonstrate how the firewall performs inspection of parsed L7 PDUs. We chose HTTP for this test. The configuration of the web server was identical to the previous tests. The firewall was configured with the default settings and the ruleset contained the following rule:

```
drop uri STR match '.*wp-admin.*';
```

The rule instructs the firewall to drop any L7 PDU, which contains a string attribute named "uri" that matches the specified regex. During the test, the client requested wp-admin.php file using the command `wget http://192.168.124.19/wp-admin.php`. The firewall logs and packet captures show that the firewall correctly dropped the HTTP request.

5.2.5 Buffer overflow

In Subsection 2.2.1 we described how an attacker may abuse a buffer overflow vulnerability in web server implementation. Our firewall can be used to prevent the exploitation of such vulnerabilities, by enforcing length limits on Application layer protocol fields. We demonstrate this functionality by setting a 20 character limit for the URI in HTTP request. This was accomplished using the following rule:

```
drop (response BOOL eq false & !(uri STR length < 20));
```

The rule rejects any HTTP request, where the URI field is at least 20 characters long. We tested the rule using the wget command, specifically `wget http://192.168.124.19/` and `wget http://192.168.124.19/a_very_long_request_uri.html`. The second requested URI is longer than 20 characters. The packet captures and firewall logs show that the firewall correctly allowed the first request but dropped the second one.

5.2.6 DNS tunneling

In Section 2.1.1, we introduced the DNS tunneling technique and the available tools. Some tunneling tools tend to use certain types of records more frequently than they occur in normal network traffic. This can be used to identify and block these tunnels. We implemented a very simple Aggregation module,[1] which can count the number of packets that satisfy a given condition, and then use these numbers to compute values and compare them. We wrote the following ruleset, which ensures that the share of DNS responses containing MX records does not exceed 20% of all DNS responses.

[1]The primary goal of this implementation is to demonstrate the functionality. To be applicable in real deployments, the module would require more features, such as grouping the counters based on certain attributes, such as source or destination IP address.

```
let agg_MX : AGG count records LIST has_item MAP has_id type NUM = 15;
let agg_DNS : AGG count src_port NUM = 53;
let 100 : AGG const 100;
let threshold : AGG const 20;
drop (src_port NUM = 53 & AGG ((agg_MX / agg_DNS) * 100) > threshold);
```

The first two lines set up the counters for the MX record responses and all DNS responses. The third and fourth lines declare named constants, which will be use used in the calculations. The last line specifies a rule, which drops any DNS response containing MX record if the threshold is exceeded. During the test, the client sent 10 DNS queries for MX and A records. The order of the queries and firewall verdicts are summarized in Table 5.1. The firewall correctly dropped DNS responses, when the share of MX record responses exceeded 20%.

# of A queries	1	2	3	4	4	4	5	6	7	8
# of MX queries	0	0	0	0	1	2	2	2	2	2
MX records share	0%	0%	0%	0%	20%	33.3%	28.5%	25%	22.2%	20%
response allowed	yes	yes	yes	yes	yes	no	no	no	no	yes

Table 5.1: Aggregation module test results

5.3 Performance testing

When designing the performance tests, we considered multiple approaches, with the first being a simulation of a real deployment. In this case, the tests would simulate the behavior of a real network, i.e. there would be multiple hosts on the network behind the firewall and a certain share of the traffic would be HTTP and DNS communication. We decided against this approach, as every network behaves differently and the results would have a low informative value. Instead, we chose to measure the round trip time (RTT), i.e. the time it takes the client machine to complete a request, in different modes of operation, e.g. with or without L7 parsing, or with the firewall completely disabled. This allowed us to easily quantify the delay introduced by the individual components involved in packet processing.

The testing environment was the same as in the previous tests. We chose HTTP over DNS for these tests as its parser is more complex and there is also overhead associated with TCP reconstruction. For measuring the RTT, we chose a tool named hyperfine.[2] The measurements were performed using the following command:

[2] `https://github.com/sharkdp/hyperfine`

```
hyperfine --runs 2000 'wget http://192.168.124.19/[request file]' \
    --export-json [result file].json --shell=none
```

The command executed the HTTP request 2000 times, each time measuring the RTT, and exported the results to a JSON file. All output files can be found in the Appendix F. The `-shell=none` option ensured that the measurements were performed without an intermediate shell, which increased accuracy.[3] We performed the tests with two files with different sizes, one being 346 B and the other 54.375 KB. For each test file, we performed the measurements in the following scenarios:

Without firewall In this test case, the firewall was not running. This measurement serves as a baseline.

No parsing In this test case, the firewall was running and filtering traffic from the forward chain, as described in Section 5.1. The firewall was configured to not parse HTTP traffic, so the incoming packets are processed as follows:

1. Parse with the L3, L4 parser.
2. Assign the packet to the correct conversation.
3. Update TCP connection state based on flags in the TCP header.
4. Since the firewall should not parse HTTP traffic, it applies the default verdict and thus allows the packet.

Allow after request In this case, the firewall buffered and parsed the HTTP request and then immediately accepted the conversation, so the rest of the conversation, i.e. the response headers and body, was allowed in a similar process as described in the previous case.

Allow after response In this case, the firewall buffered, parsed and allowed the HTTP request. Then it waited for, buffered and parsed the HTTP response headers and then allowed the rest of the conversation[4] to pass through the firewall without any additional L7 parsing. The purpose of this test case is to observe how the buffering and examining of the response header affects the performance.

The modifications in firewall configuration files used create these scenarios are documented in the electronic attachment. We repeated the measurements for each test file and each of the test cases described above four times, to ensure they yielded consistent results. The average values are presented in the figure below.

[3] `https://github.com/sharkdp/hyperfine?tab=readme-ov-file#intermediate-shell`

[4] In this case the response body.

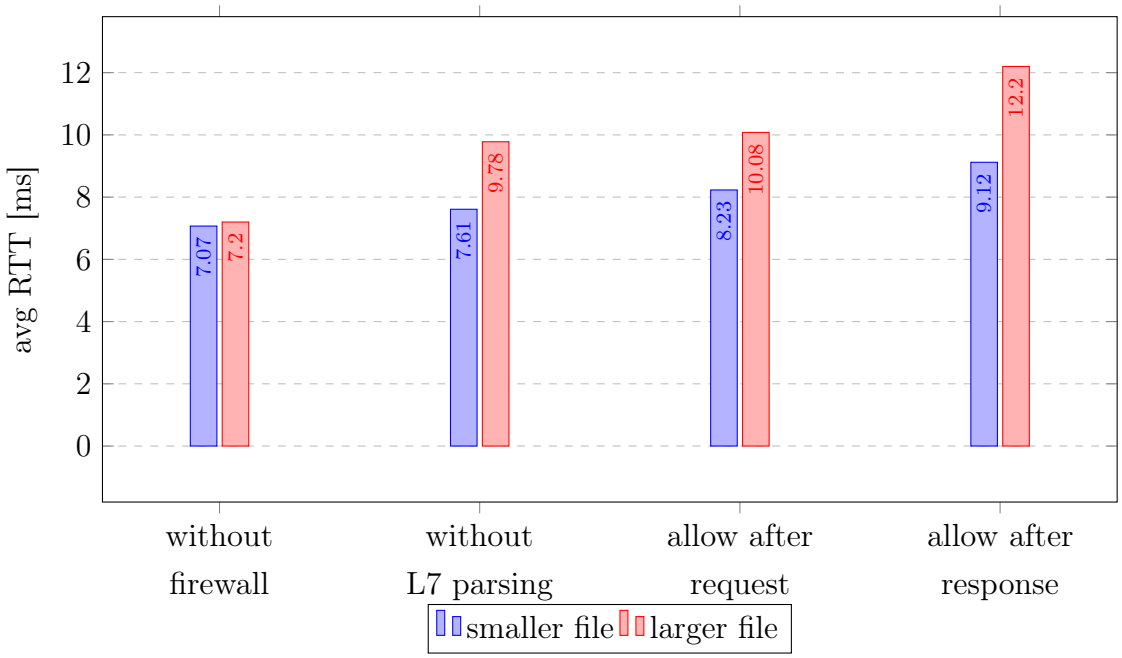

Figure 5.2: Performance results

As we expected, the increasing complexity of the traffic inspection resulted in higher RTT values. We can also observe that the increase is faster in tests with the larger file. This may be caused by the increase of packet RTT (not the RTT we measure in these tests) caused by the firewall interfering with the TCP congestion control mechanism. For reasons explained in Section 6, confirming this is out of scope of this thesis.

	no firewall	no L7 parsing	allow after request	allow after response
small file	100%	107.64%	116.4%	128.99%
large file	100%	135.83%	140%	169.4%

Table 5.2: Percentage increase in RTT

5.3.1 Further analysis of results

If we look at the average RTT value for the smaller file without the firewall and L7 parsing, we can see that the difference is 0.54ms. In the first test case, the packets are processed only by the kernel. In the second case, with the firewall enabled, the packet is copied between kernel-space and user-space, the L3 and L4 headers are parsed and the TCP connection status is updated. As we can see, this introduces a sub-millisecond delay. In the case of the larger file, the difference between the first two tests is larger. Using tcpdump, we counted the number of packets in a TCP connection,[5] which transferred

[5]All packets in a connection including three-way handshake, acknowledgements, etc.

the smaller file and the larger file, and calculated the delay increase per packet. This should yield similar values, as each packet undergoes the same processing.

	no firewall	no L7 parsing	RTT diff	# of packets	delay per packet
smaller file	7.07ms	7.61ms	0.54ms	10	0.054ms
larger file	7.2ms	9.78ms	2.58ms	21	0.123ms

Table 5.3: Delay per packet calculation

As we can see, the delay per packet values are significantly different. This could be caused by inaccuracy in measurements, as we are examining sub-millisecond values, or the inherent delay associated with the operation of the firewall interferes with the TCP congestion control mechanism and decreases the throughput of the TCP connection. Further investigation and a more accurate testing environment are necessary to identify the cause of this difference.

5.3.2 CPU usage

Besides the RTT, we also measured the CPU usage. These measurements were performed separately, to eliminate possible interference. In the tables below, we summarize the CPU usage values during the measurements. We list only the user and system values, as io-wait[6] was negligible and steal[7] irrelevant. First, we performed the measurements with the firewall disabled and idle.[8] The values listed in Table 5.4 are the averages calculated by command `sar 1 60`.[9] The increase in CPU usage was negligible. Table

	disabled	enabled
user	0.88%	1.17%
system	1.24%	1.19%

Table 5.4: Disabled and idle CPU usage

5.5 summarizes the average CPU usage during the different performance test cases. Each value is an average calculated by command `sar 1 25`. During the measurement, we ran the aforementioned hyperfine command, but with `-runs` flag set to 5000 to ensure we can collect enough CPU usage samples. As expected, the firewall increases the CPU usage. We can approximate the CPU usage of the firewall by comparing the measured values when the firewall was disabled, the first column, and when it actively filtered the network traffic, the second and third columns. The difference between the

[6]The percentage of time spent waiting for I/O operations.

[7]The percentage of time the virtual CPU spends involuntarily waiting.

[8]The firewall was running, but we were not actively sending any traffic as in the other tests.

[9]The command measured the CPU usage 60 times and waited 1 second between the measurements.

file + measured values		test case			
test file	values	no firewall	no parse	allow request	allow response
small file	user	25.54%	25.33%	28.18%	27.3%
	system	8.44%	8.95%	6.21%	6.15%
	system + user	33.98%	34.28%	34.39%	33.45%
large file	user	26.57%	30.56%	30.68%	30.81%
	system	8.27%	11.41%	11.68%	9.54%
	system + user	34.84%	41.97%	42.36%	40.35%

Table 5.5: CPU usage during performance tests

tests with the small file did not exceed 1% and with the large file, it was approximately 7.5%. An interesting observation is the slight drop in CPU usage between the last two test cases. This can be explained by the necessary wait between sending the processed HTTP request and receiving the response.

5.3.3 Performance issue

We attempted to test how the firewall handles parsing the entire TCP connection, i.e. everything from the first packet after the three-way handshake to the last ACK packet, is processed by the parser threads. However, we were not able to measure reliable data for this test case. We observed a significant drop in performance, which did not occur during every test run and in case it occurred, its severity differed. In the worst observed cases, the RTT values reached from 0.2 to 0.8 second. An example hyperfine JSON is included in Appendix F. We also observed, that in this situation, the CPU usage significantly decreases. This can be seen in sar command output also included in the Appendix. This indicates, that the issue is likely caused by a component in the firewall needlessly halting. We attempted to identify the cause, by analyzing the possibly problematic elements of the implementation, such as the NFQUEUE library workaround that uses signals, or possible decrease in TCP connection throughput caused by the buffering delay interfering with the TCP congestion control mechanism.[10] Based on our observations, the most likely cause is in the NFQUEUE library workaround, as we observed similar delays when issuing other signals to the process, such as SIGTERM, which gracefully shuts down the firewall. We tried to replicate the issue outside the firewall, but this was unsuccessful. Further testing is necessary to determine the cause of this issue. However, this is out of the scope of this thesis, as it requires more complex tests and testing environment, including a special TCP/IP stack implementation, and possibly analyzing the Linux kernel source code.

[10]The cause of TCP congestion control problem is discussed in more detail in Chapter 6.

Chapter 6

Further work

Testing Testing is an important aspect of software development, especially in the case of security software. As a part of this thesis, we wrote unit tests for chosen components, e.g. the rule management module or parsers, but thorough testing is out of the scope of this thesis. For example, consider the TCP congestion window solution. To thoroughly test it, we would need a testing environment, which allows fine-grained control over the progress of the TCP connection. There are traffic generators available, such as nping [30], which can configure certain parameters of the TCP connection, such as window size, but this is not sufficient to create test scenarios for the discussed feature. Ultimately, we would have to implement a TCP/IP stack, which allows fine-grained control over the progress of the TCP connection. This is out of the scope of this thesis.

Issues discovered during development During the implementation and testing phase, we identified another potential issue with TCP. The firewall can be configured to buffer TCP segments and release them only when a valid Application layer PDU can be reconstructed. This mode of operation is useful when we want to allow the conversation to flow, but still analyze it. The buffering increases the round trip time (RTT) value used in the TCP congestion control mechanism. This artificial increase can cause the communicating hosts to assume that the network is congested and change their behavior accordingly. We did not explore or address this for similar reasons as described in the previous point. Further testing is necessary to determine the impact, and alternative approaches, such as proxying the TCP connections,[1] should be explored.

Further functionality There are many possibilities to expand the current functionality. For example, HTTP/1.1 is a complex protocol and implementing support for

[1]The firewall would have two separate TCP connections, one with client and one with server. This may alleviate potential issues caused by the current approach.

all its features in our parser was out of the scope of this thesis. Other examples include adding support for more Application layer protocols, command-line rule editing utility,[2] more configuration options or better granularity, more intuitive configuration grammar, more verbose logs etc.

[2]Similar to iptables.

Conclusion

Through designing, implementing and testing, we investigated the feasibility of creating a purely software L7 firewall that runs in the user-space of OS Linux. Regarding the L7 protocol analysis, we focused on analyzing HTTP and DNS, as they are essential for the operation of the Internet and a potential attacker may abuse these protocols to perform an attack or bypass insufficient defense mechanisms.

During implementation, we discovered multiple issues, which had to be addressed. The buffering approach in combination with TCP proved to be the most problematic aspect of the design. Several problems stemmed from this scenario, for which we proposed and implemented solutions. However, there still remain unanswered questions regarding the TCP congestion control mechanism and how it affects the TCP connection throughput. Proper verification requires further complex testing. Exploring alternative approaches, such as proxying the TCP connection should be done, as it may lead to a cleaner and more efficient design.

The test results demonstrate that the implementation is capable of validating and analyzing L7 protocols and subsequently filtering based on rulesets of varying complexity. The tests also present how the advanced functionality aids in defending against L7 protocol abuse. The performance tests showed the CPU usage and latency increase are within acceptable margins.

Overall, the findings and the implementation itself appear promising, and they can be used for further research into this topic and the development of user-space firewalls with advanced filtering capabilities.

Bibliography

[1] Kenneth Ingham, Stephanie Forrest, et al. A history and survey of network firewalls. *University of New Mexico, Tech. Rep*, 2002.

[2] netfilter/iptables project homepage — The netfilter.org project. `https://netfilter.org/`. Retrieved December 3, 2023.

[3] Service Name and Transport Protocol Port Number Registry. `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml`. Retrieved December 12, 2023.

[4] Cisco secure firewall 3100 series data sheet. `https://www.cisco.com/c/en/us/products/collateral/security/firewalls/secure-firewall-3100-series-ds.html`. Retrieved November 19, 2023.

[5] Next-generation firewall hardware. `https://www.paloaltonetworks.com/network-security/next-generation-firewall-hardware`. Retrieved November 19, 2023.

[6] Hardware acceleration | FortiGate | FortiOS 7.4.1 | Fortinet Document Library. `https://docs.fortinet.com/document/fortigate/7.4.1/hardware-acceleration/448300/hardware-acceleration`. Retrieved December 11, 2023.

[7] Cloudflare — Cloud-Based WAF Security. `https://www.cloudflare.com/application-services/products/waf/`. Retrieved December 15, 2023.

[8] SpiderLabs/ModSecurity. `https://github.com/SpiderLabs/ModSecurity`, December 2023. Retrieved December 15, 2023.

[9] Snort — Network Intrusion Detection & Prevention System. `https://www.snort.org/`. Retrieved December 11, 2023.

[10] Suricata. `https://suricata.io/`. Retrieved December 11, 2023.

[11] Suricata 8.0.0-dev documentation — 8.45. differences from snort. https://docs.suricata.io/en/latest/rules/differences-from-snort.html. Retrieved December 11, 2023.

[12] Romain Fouchereau. 2022 global dns threat report securing anywhere networking — efficient ip. https://efficientip.com/wp-content/uploads/2022/10/IDC-EUR149048522-EfficientIP-infobrief_FINAL.pdf, Jun 2022. Retrieved December 15, 2023.

[13] Ron Bowes. iagox86/dnscat2. https://github.com/iagox86/dnscat2. Retrieved January 16, 2024.

[14] Erik Ekman. yarrick/iodine. https://github.com/yarrick/iodine. Retrieved January 16, 2024.

[15] Radim Koza. *Real-time Detection of Network Tunnels*. Bachelor's thesis, Masaryk University, 2017.

[16] Man page of iptables-extensions. https://ipset.netfilter.org/iptables-extensions.man.html. Retrieved May 19, 2024.

[17] Let's Encrypt Stats. https://letsencrypt.org/stats/#percent-pageloads. Retrieved March 16, 2024.

[18] HTTPS encryption on the web — Google Transparency Report. https://transparencyreport.google.com/https/overview?hl=en. Retrieved March 16, 2024.

[19] Cisco firepower Management Center Configuration Guide, Version 7.0 — Understanding Traffic Decryption. https://www.cisco.com/c/en/us/td/docs/security/firepower/70/configuration/guide/fpmc-config-guide-v70/understanding_traffic_decryption.html#concept_E2E5B846EA6A4C8ABD963BE167899315. Retrieved March 16, 2024.

[20] Dominic Lovell. HTTP | 2021 | The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2021/http, 2021. Retrieved March 16, 2024.

[21] Commonly Open Ports. https://www.speedguide.net/ports_common.php. Retrieved March 16, 2024.

[22] libnetfilter queue — documentation. https://netfilter.org/projects/libnetfilter_queue/doxygen/html/. Retrieved March 8, 2024.

[23] C VS Rust benchmarks, Which programming language or compiler is faster. `https://programming-language-benchmarks.vercel.app/c-vs-rust`. Retrieved February 18, 2024.

[24] What is Ownership? — The Rust Programming Language. `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`. Retrieved March 8, 2024.

[25] Gavin Thomas. A proactive approach to more secure code | MSRC Blog | Microsoft Security Response Center. `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/`. Retrieved January 8, 2024.

[26] The Chromium Projects — Memory safety. `https://www.chromium.org/Home/chromium-security/memory-safety/`. Retrieved January 8, 2024.

[27] std::option — Rust. `https://doc.rust-lang.org/std/option/`. Retrieved May 19, 2024.

[28] 27.4.3.2. Parsers — Suricata 8.0.0-dev documentation. `https://docs.suricata.io/en/latest/devguide/extending/app-layer/parser.html`. Retrieved February 18, 2024.

[29] Paul Mockapetris. Domain names — implementation and specification. RFC 1035, RFC Editor, November 1987. Section 4.1.4.

[30] nping — Linux manual page. `https://man7.org/linux/man-pages/man1/nping.1.html#TCP_MODE`. Retrieved April 13, 2024.

Appendix A

Firewall source code

The electronic attachment contains the source code of the firewall implementation described in this thesis. Instructions for use are included in a README.txt file. Configuration file grammar is described in Appendix E and the parser output specification, i.e. the keys and value types associated with Application layer protocols, can be found in Appendix D.

Appendix B

User-space latency penalty

The program we used for testing is included as an electronic attachment. The program starts by binding to NFQUEUE queue number 0 and then accepts all received packets in an infinite loop. To redirect traffic to this program, we manually added the following iptables rule:

```
iptables -A INPUT -i [iface] -j NFQUEUE --queue-num 0 --queue-bypass
```

The test was performed in a virtual machine running Fedora Linux with 2 virtual CPUs and 4 GB of memory. `[iface]` was substituted by the name of the network interface of the virtual machine. The latency was measured using the ping command on the physical host and results are included in the form of CSV files as an electronic attachment.

Appendix C

Threading test

The program we used for testing is included as an electronic attachment. Its purpose is to spawn and run a large amount of consumer and one producer thread. The producer sends a large amount of messages, each time choosing a random consumer. However, this does not simulate the behavior of the firewall accurately. The speed at which the producer can send the messages is higher than what it would be in a real environment. This is because the maximum number of packets that can be processed by a network interface card at 1 Gbps speed is smaller than what a thread without any I/O interactions can produce. The real number of packets would be even smaller because not all Application layer protocols are parsed by our firewall, so some packets a returned with default verdict and do not cross any thread boundaries.

In this test, we did not aim to accurately simulate the behavior of the firewall as the number of processed packets largely depends on the network, where the firewall is deployed, and its configuration. We intended to explore how the thread management overhead affects CPU usage. We measured this in two scenarios:

- producer running at full speed
- producer sleeping for 1ms[1] between sending messages with a given probability

Measurements

All tests were done in a virtual machine running Fedora Linux with 2 virtual CPUs and 4 GB of memory. We measured the CPU usage every second during a 40-second interval using the sar command. Full sar command outputs can be found in the electronic attachment. Here we present the measured values in the form of graphs. The graph for each test shows the CPU usage during the test and the baseline measurement. There are

[1]Accuracy of the sleep method depends on the operating system, so this may introduce deviations in the measurements. However, this is not a concern, as we are interested only in estimates.

also two gray dashed vertical lines. The first one marks the moment when all threads were spawned and the second one when they all terminated.

No delay

Parameters:

consumer thread count: 10000

message count: 2000000

probability of delay: 0%

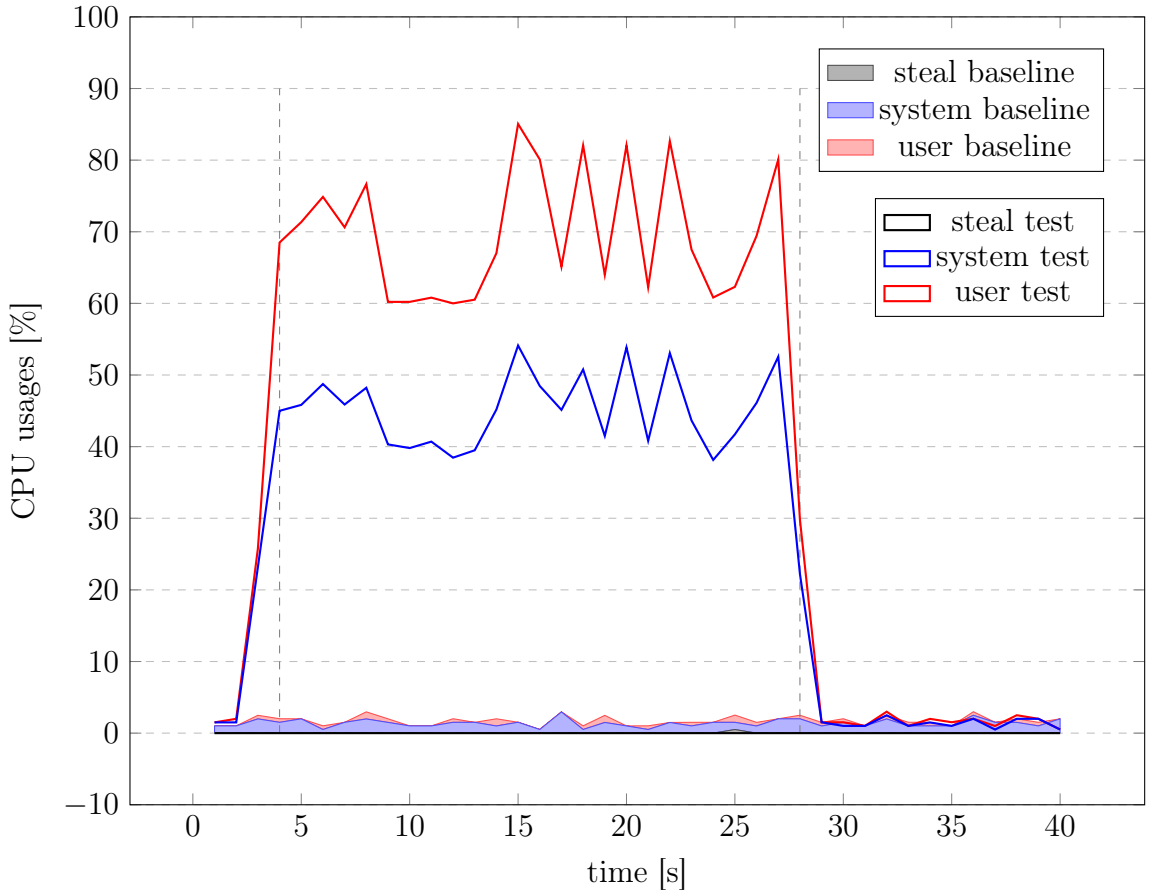

Figure C.1: Thread test — full speed

Delay with 10% probability

Parameters:

consumer thread count: 10000

message count: 150000

probability of delay: 10%

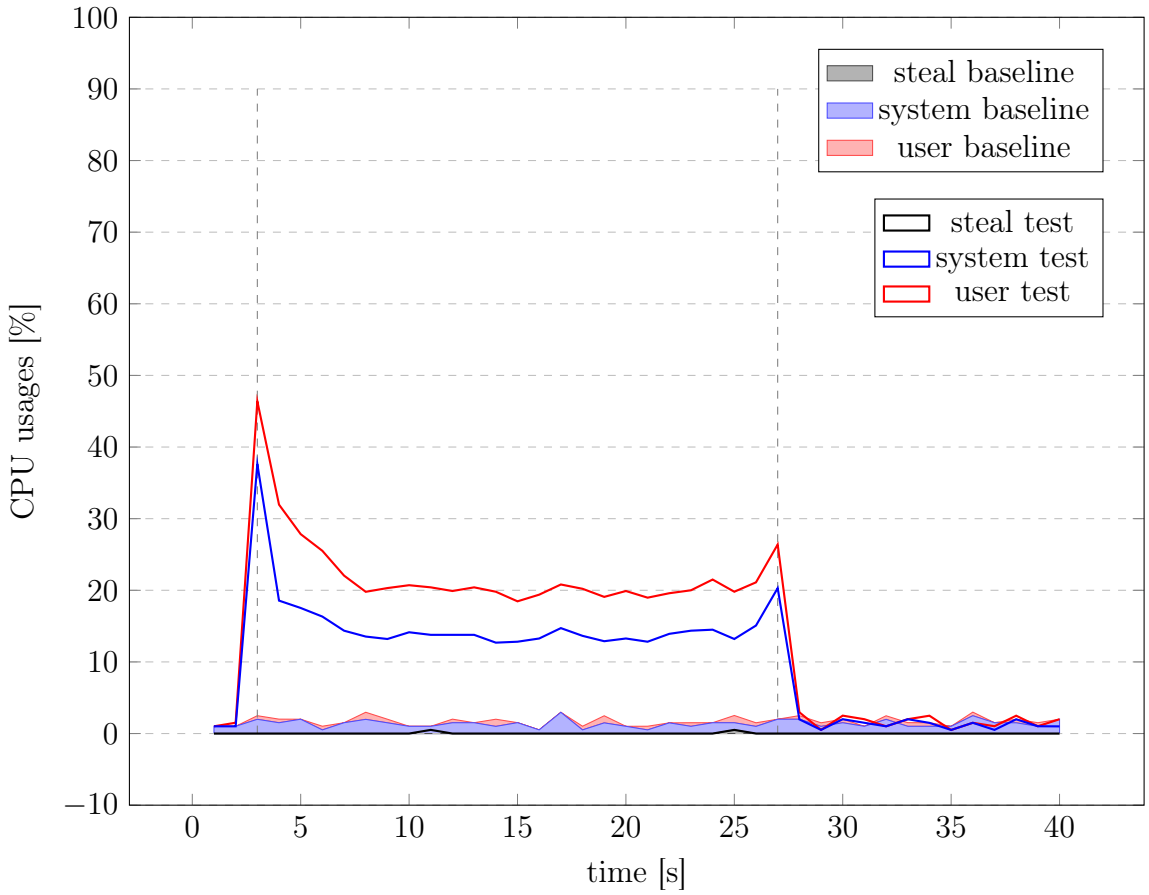

Figure C.2: Thread test — with delay

Appendix D

Parser output specification

The HTTP and DNS parsers return key-value pairs that represent a parsed L7 PDU and can be used to define rules. We will now list the possible keys with associated value types and describe the semantics.

D.1 Common attributes

There are some attributes, which are included in every parsed L7 PDU, regardless of the Application layer protocol.

key	**value type**	**description**
src_ip	IP address	the IP address of the sender of the PDU
dst_ip	IP address	the IP address of the receiver of the PDU
src_port	number	transport layer source port
dst_port	number	transport layer destination port

Table D.1: Common parsed PDU attributes

D.2 HTTP

All HTTP messages have the following attribute.

key	**value type**	**description**
response	bool	true if the message is HTTP response

Table D.2: Mandatory HTTP message attributes

The following attributes are associated with the individual HTTP message types.

key	**value type**	**description**
method	string	mandatory in HTTP request
uri	string	mandatory in HTTP request
host	string	mandatory HTTP request header

Table D.3: HTTP request specific attributes

key	**value type**	**description**
return_code	number	present if the message is HTTP response
response_code_word	string	present if the message is HTTP response
response_body	bytes	present if the response is HTTP response, empty if there is no response body

Table D.4: HTTP response specific attributes

All HTTP messages can contain optional header fields, which comprise header field name and header field value separated by a colon. These are included as attributes in the following format:

key	**value type**
«header_field_name»	string

Table D.5: HTTP header field syntax

HTTP header field names are case-insensitive, so the parser converts them to lowercase. The header field values can include comments or multiple delimited values. These cases are not parsed, but rather the entire value string is included in the key-value map as is. If a rule has to account for this, the regex operation is a suitable tool.

D.3 DNS

key	value type	description
id	number	ID of the DNS message
response	bool	true if the message is a DNS response, this is determined based on the "direction" of the message
opcode	number	OPCODE field from the DNS message
aa	bool	Authoritative answer field from DNS message
tc	bool	Truncation field from DNS message
rd	bool	Recursion desired field from DNS message
ra	bool	Recursion available field from DNS message
z	bool	Zero field from DNS message
rcode	bytes	Response code from DNS message
questions	list	list of all entries in the questions section
records	list	list of all entries in the answer section
name_servers	list	list of all entries in the name server section
additional_records	list	list of all entries in the additional records section

Table D.6: DNS header attributes

The last four values are lists of maps. Each map represents a question or records using a predefined set of attributes.

key	value type	description
class	number	class code
type	number	requested record type, e.g. A, MX, etc.
name	string	resource (domain) name

Table D.7: DNS question attributes

The following attributes are used for all three record types.

key	value type	description
class	number	class code
type	number	record type
name	string	resource (domain) type
ttl	number	record time to live in seconds
rdata	bytes	additional record-specific data, which is not further parsed

Table D.8: DNS record attributes

Appendix E

Configuration file syntax

We will now describe the configuration file syntax in the form of a grammar in ABNF. The basic semantics and how the configuration is used by the firewall are described in Section 4.3. This appendix focuses on the formal specification.

E.0.1 Basic definitions

Before we describe the configuration syntax, we will define common productions, which will be frequently referenced.

```
ASCII_DIGIT        = '0'..'9' ; digits 0 to 9
ASCII_ALPHA        = 'a'..'z' | 'A'..'Z' ; all lower and uppercase
                                           alphabetical characters
ASCII_ALPHANUMERIC = ASCII_DIGIT | ASCII_ALPHA
WHITE_SPACE        = " " | "\n" | "\r\n" | "\r"
ASCII_HEX_DIGIT    = '0'..'9' | 'a'..'f' | 'A'..'F'
id                 = (ASCII_ALPHANUMERIC | "-" | "_")+
number             = ASCII_DIGIT+
quoted_str         = "'" ANY* "'" ; ANY - any character
                                    except the single quote
cmp_op             = "<" | ">" | "=" | "<=" | ">="
```

E.0.2 Configuration file

The configuration file consists of two types of statements — *rules* and *declarations*. Each statement has to be terminated by a semicolon, which can be followed by arbitrary white space characters.

```
configuration = ( (rule | declaration) ";" WHITE_SPACE* )*
```

E.0.3 Condition

Before we describe the two types of statements, we will introduce a *condition* production, which is used by both statement types.

```
negated_condition = "!(" condition ")"
binary_condition  = "(" condition ("&" | "|") condition ")"
atomic_condition  = id type_specific_condition
condition         = atomic_condition | binary_condition |
                    negated_condition | aggregation_condition
```

The aggregation_condition will be defined later together with the aggregation module declarations, as they are closely related. We will now have a closer look at the *atomic_ condition* production. As mentioned in the Subsection 4.3.1, the atomic_condition specifies conditions for information returned by an L7 parser. In the case of successfully parsed PDU, the parser returns a key-value map, where the key is a string and the value can be one of the following types:

- string
- number (Rust usize type)
- boolean
- IP address
- byte array
- list
- map with string keys

The last two types can contain values of any type, e.g. a valid list value could be {false, 42, "test", {123, true}, 1}. Each type has associated conditions that can be performed.

```
type_specific_condition = (num_op | str_op |
                           bool_op | ip_op |
                           bytes_op | list_op |
                           map_op)
```

The atomic condition is evaluated as follows:

1. Get the value from L7 parser output specified by the id in atomic_condition, the result is false if the key does not exist.
2. Check if the value type matches the one required by the type_specific_condition, the result is false if the types do not match.
3. Evaluate the type_specific_condition, return the result.

We will now describe the individual type_specific_conditions. The type for each condition is specified by an uppercase literal.

Number and string types

```
num_op = "NUM " cmp_op number
str_op = "STR " (("eq " | "match " | "contains " ) quoted_str |
                 "length < " number)
```

The semantics of number conditions are apparent. The string conditions check the following:

- eq – equality between examined string and the quoted_str
- match – the quoted_str is interpreted as regex and the rule checks if it matches the examined string
- contains – checks if the examined string contains the quoted_str
- length < – checks if the examined string is shorter than the value specified by the number

IP address type

```
ip_addr = (ASCII_DIGIT | ".")+ | (HEX_DIGIT | ":")+
ip_op   = "IP " ("eq " ip_addr |
                 "in " ip_addr "/" number |
                 "in list " id)
```

The IP address conditions use a production named ip_addr. It does not match only IP addresses, but any string containing decimal or hexadecimal digits with dots or colons. Whether these strings are valid IP addresses is checked during the creation of condition structures. The three possible operators have the following semantics:

- eq – Checks for equality between the IP addresses.
- in – Checks if the examined IP address is within the subnet specified by IP address and mask in CIDR notation following the operator. The validity of the mask is checked during the creation of condition structures.
- in list – The operator is followed by an id, which specifies a list of IP addresses stored in the IP addresses list module. The condition is true if the evaluated IP address is in the specified list.

Byte array type

```
bytes_op = "BYTES " ("pattern "(HEX_DIGIT HEX_DIGIT)+ |
                     "length < " number)
```

The first type of byte array condition contains a byte pattern specified as pairs of hexadecimal digits. This condition is true when the examined byte array contains the pattern. The second type is similar to the length condition for string — it checks if the length of the byte array is shorter than the number.

List and map types

```
list_op = "LIST " type_specific_condition
map_op = "MAP " id " " type_specific_condition
```

The list and map conditions both have a type_specific_condition as an argument. They evaluate as true, when:

- list – there is at least one item in the list, which satisfies the condition
- map – the value that belongs to the key specified as id satisfies the condition in the argument

E.0.4 Rule

The syntax of a rule statement consists of a verdict and a condition:

```
rule = ("allow" | "drop" | "allow pdu") condition
```

The semantics are described in Section 4.5.3. 4.3.1.

E.0.5 Declaration

Each declaration starts with the keyword let and an id:

```
declaration = "let " id " : "
```

There are three types of declarations — one for each module.

IP address lists

The IP address lists is the simplest module. Its declaration consists of a keyword and a comma-delimited list of IP addresses:

```
list_decl = "LIST " (ip_addr ", ")* ip_addr
```

Rate limit

The Rate limit module limits the number of L7 PDUs that match a specified condition and can pass through the firewall. The declaration consists of a keyword, the number of packets that are allowed through the firewall in a given number of seconds and the condition:

```
rate_lim_decl = "RATE " number "/" number "s " condition
```

Aggregation

The aggregation module stores numbers together with their IDs. The numbers can either be constant or they can count how many packets satisfy a given condition:

```
agg_type = "count"
agg_decl = "AGG " ("const " number | agg_type " " condition)
```

We will now define the aggregation_condition, which was omitted in Section E.0.3.

```
atomic_expr = id
arithm_op             = "+" | "-" | "*" | "/"
bin_expr              = "(" expr " " arithm_op " " expr ")"
expr                  = bin_expr | atomic_expr
aggregation_condition = "AGG " expr " " cmp_op " " expr
```

The aggregation_condition compares two arithmetic expressions, which consist of IDs, that serve as variables and arithmetic operators. When evaluating a condition like this, the values of the expressions are calculated using the numbers stored in the Aggregation module belonging to the IDs. The two resulting values are then compared to obtain the final result.

Appendix F

Test results

The DNS and web server configuration files, packet captures and logs produced during the tests are included as an electronic attachment.