Firewall simulator as a WebApp

Patrik Mjartan

4th Year Project Report
Computer Science and Management
School of Informatics
University of Edinburgh

2017

Abstract

Firewall is a rather straightforward entity in its core - packets trying to get through get inspected and are either let through, or denied. However, configuring and testing a firewall setup can be found very inaccessible to most people. That is, one would have to use at least two machines in order to configure and test his setup. This in itself can take several hours and acts as a huge discouragement to most students.

This project sought out to create a firewall simulator as a WebApp, hence erasing the potentially difficult and time consuming act of setting up the machines in order to test our firewall ruleset.

There are two terminals, at the WebApp, that try to simulate linux terminal. Users can setup their firewall rules in both terminals and see what behaviour it accomplishes. For example, in one terminal, we can block all packets coming to our port 22 and we can test whether this works using the other terminal.

Acknowledgements

Foremost, I would like to thank my parents and my brother for enabling me to pursue my studies abroad. Without their financial and mental support I would most certainly not be able to study at this university.

I would also like to thank William Waites and Kami Vaniea for allowing me to make an adjust the original project - `Firewall administration, the game` into something that I thought would be more interesting - `Firewall simulator as a WebApp`.

Lastly, I would like to thank Adam Pavlisin for being patient while I spammed him with my JS/CSS/Webdev enquiries.

Table of Contents

Chapter 1

Introduction and Background

Firewall is a piece of software (or hardware) that, if correctly setup, can protect our network from being *successfully* attacked over the internet (or from within the network!). Hence there is a need to have a correctly setup firewall.

This project is split into 4 core chapters:

chapter 1 Detailing the background knowledge that needs to be understood before understanding the future chapters; along with showcasing some already existing games and how I thought I could contribute.

chapter 2 Detailing my goals for the project in more details, what I required to carry out these tasks, what I did, how I tested it to make sure it behaves as expected and finally reflecting on my design and implementation.

chapter 3 Detailing whether the users and the experts think the project is usable and suitable for educational purposes. Basically, whether what I made sense and whether they could see themselves using it.

chapter 4 Detailing my thought at the end of project, discussing what I would have done differently have I started again and the potential extensions and additions to the system.

In this chapter we take a retrospective look at what had to be understood before undertaking this project. First of all, we must understand how firewall works (section 1.1) before we can attempt to look at some already existing games that deal with firewall. This includes looking at how TCP/IP internet protocol works (subsection 1.1.1) and how packets are filtered (subsection 1.1.2). Then we can have a look at the already existing games (section 1.2). Next, I will explain the reasoning why I decided to contribute with what I did (section 1.3) and lastly, we look at `iptables` (section 1.4) as this is the system I was re-implementing in the project.

1.1 How a firewall works

This section describes how TCP/IP packages its information, and how firewalls decide to allow or deny traffic. TCP/IP traffic is broken into packets, and firewalls must examine each packet to determine whether to drop it or forward it to the destination. [31]

1.1.1 How a TCP/IP protocol works

This section will attempt to explain how TCP/IP protocol [10] works and why it is important to understand in order to understand how a firewall works.

In general, when we want to develop an application, it is not one man's job and hence there are multiple developers working in union. Ideally, developers' jobs will not overlap too much, that is, for instance the team working on UI will not change the code of the business logic (which is a responsibility of another team). However, they will need to rely on the business logic team to provide a way for the UI team to get the information needed to be displayed in the UI. That is, UI team simply wants to call method `getPrice()` and does not really care how it is coded, as long as it produces the results they need.

This abstraction of work is called *encapsulation* and it is an important aspect of how TCP/IP protocol works. TCP/IP consists of four layers, namely [10]:

- Application
- Transport
- Internet
- Link

Each layer abstracts how it works to the layer above, in the same sense how the UI developer does not need to know how `getPrice()` works, a developer working on Transport layer does not need to know how Internet layer works.

The `application` layer creates data that is to be transmitted to other application - *process* - on the same or, usually, another host. To do this, the application uses the services provided by Transport layer. Application layer is where protocols such as SSH, HTTP or FTP are made. In essence application (process) tells the transport layer to send a chunk of data (packet) to a certain process on a certain host using a certain protocol (TCP or UDP).

The `transport` layer takes care of the host to host communication. It takes the packet and sends it to the host that the application layer said we should send it to. It sends it using either UDP or TCP protocols (application layer specifies which).

- TCP provides a one-to-one, connection-oriented, reliable communications service. [20]
- UDP provides a one-to-one or one-to-many, connectionless, unreliable communications service. [20]

In essence, we use TCP when we must ensure that the connection is very 'important', that is, sensitive or important information is being send. Examples would be online banking or account management. UDP, on the other hand, is used when we only really care about the speed and TCP's overhead (of ensuring everything is received) is too big. This is used in services which do not need to ensure that everything will be received. Examples would be video streaming (e.g. we do not really care if some particular pixels are lost) or video games.

The `internet` layer is responsible for addressing, packaging, and routing functions. [32]

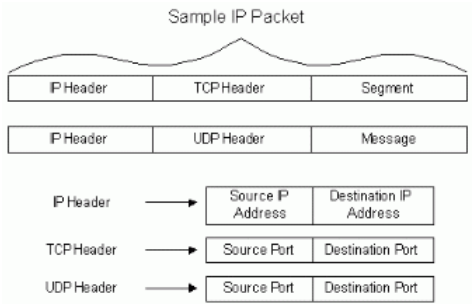


Figure 1.1: Simplified packet structure

The `network` layer (or `Network Access` layer) is responsible for placing and receiving TCP/IP packets top and from the network. TCP/IP was designed to be independent of the network access method. [32]

So why is this important in order to understand how firewall works? The TCP/IP traffic is broken into packets (`application` layer) and the firewalls examine whether each packet is dropped or forwarded (allowed) farther. This process takes place in both the outgoing application, but more importantly in the incoming application.

1.1.2 How packet filtering works

After having a brief understanding of TCP/IP protocol, let us look at how a packet is structured at Figure 1.1.

As you can see, there are three parts:

- IP Header - contains source (sender) and destination (receiver) IP addresses
- TCP/UDP Header - contains the source and destination ports. Each process (application) that needs to communicate with other process is assigned a port. This defines the location for delivery of the packet on the given receiver host.
- Segment/Message - contains actual content, the message we want to send to a different process.

Let us take a look at a practical example to gain better understanding. When we go to `google.com`, our browser (process) sends an HTTP request to google's webserver, and the request contains the following:

- IP Header - our IP; google's webserver IP
- TCP/UDP Header - our browser's port; google's webserver port
- Segment/Message - not relevant

Our browser's port identifies which application (process) sent the request. This is mostly useful for one thing - when we get a response, where should our machine (host) send it (ie. to what process)? Well, to the port it originally came from - our browser. In this case, google's web-

server would swap around the port numbers in its response, ie. now the destination port is our browser's port, and the source port is the google's webserver port.

The primary purpose of a firewall is to filter traffic. Firewalls inspect packets as they pass through, and based on the rules we defined, the firewall allows or denies each packet. [31]

There are at least two firewalls in our example:

Hence, there are at least two places where our packet can be blocked (out firewall; and Google's firewall).

We use packet filters to tell firewall to drop traffic that meets specific criteria. For instance, we could make a rule (filter) that would drop all requests coming to our web server; although that might be a bad idea, as we probably want our web server to serve our content to whoever visits our website. However, we can also filter based on an IP rule, IP protocol and so forth. These are the attributes we can filter on in our filtering rules [31]:

- Source IP
- Destination IP
- IP protocol (UDP/TCP)
- Source port
- Destination port
- Interface from where the packet arrived
- Interface where the packet is destined

We could, for instance, make a rule that will block requests headed to google specifically, and so now the packet would leave our browser, but would be blocked by our firewall:

Similarly, google could have a rule that would block requests coming from our IP address:

1.2 Existing games

After gaining a better understanding on firewalls and TCP/IP protocol, we can look at some already existing games that deal with security.

1.2.1 Seed Labs

In this subsection we will look at some of the seed labs [12], more specifically, network security seed labs [22].

1.2.1.1 TCP/IP Attack Lab

This subsection deals with *TCP/IP Attack Lab* [23] and the ideas are hence based upon the said lab.

This lab's objective is for people to gain experience on vulnerabilities and attacks exploiting these vulnerabilities.

"Wise people learn from mistakes. In security education, we study mistakes that lead to software vulnerabilities. Studying mistakes from the past not only help students understand why systems are vulnerable, why a "seemly-benign" mistake can turn into a disaster, and why many security mechanisms are needed." [23]

Furthermore, it also helps to teach the common patterns of vulnerabilities so we can prevent being exploited by them in future. [23] Using vulnerabilities as case studies, people can learn how to improve in developing secure design, how to program more securely, and how to test whether our products are secure. [23]

To conduct this lab, students need to have at least 3 machines. One computer is used for attacking, the second computer is used as the victim, and the third computer is used as the observer. [23]

When I was doing this lab, this was the point where I realised that although the topic is interesting, it is a lot of work to setup the environment (at least compared to other areas that require only a single machine). Sure, we can install 3 virtual machines to simulate the environment, but that requires a lot of effort (for someone who has no experience with virtual machines) that could be used on the learning objective of the lab instead. Anyway, let us dive further into the lab to see what it is actually about, assuming we could fulfil the 3 machines requirement.

We have to setup the three machines in a way that they will have IPs:

- 192.168.0.122
- 192.168.0.123
- 192.168.0.124

Where our gateway IP is 192.168.0.1. This can be also seen at Figure 1.2.

As you can see, all of these IPs start with 192.168, which you might think is quite uncommon, as the IPs are assigned by your internet service provider (ISP) and surely the chances that 4 devices on our network all are so similar is slim.

What happened here however, was that internet was designed in times where it was not expected that we will ever need more than 2^{32} IP addresses (each IP consists of 4 sets of bytes, e.g. 0.0.0.0 would actually be 00000000 00000000 00000000 00000000 and 1.1.1.1 would be 11111111

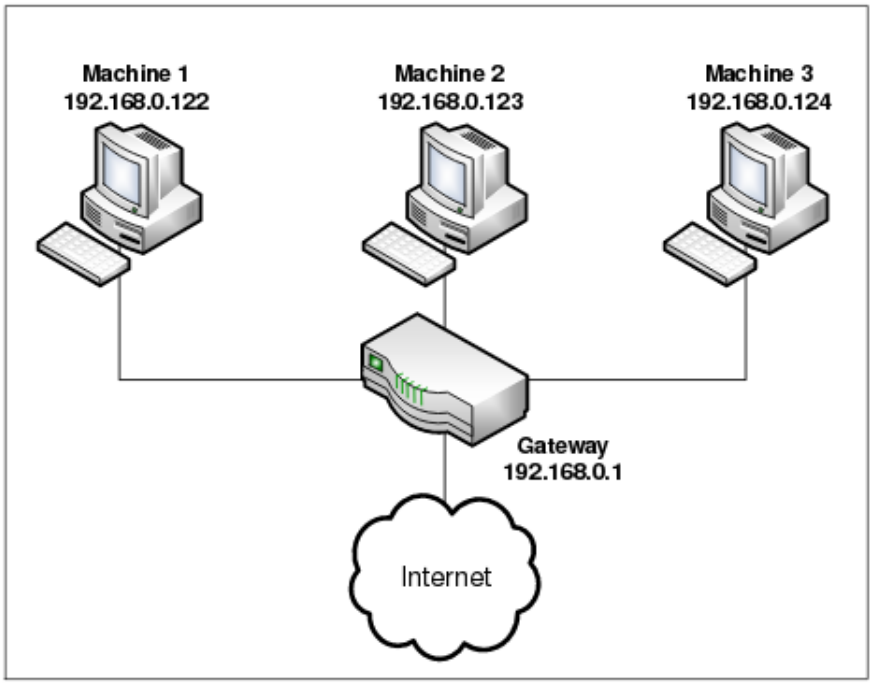


Figure 1.2: Environment setup, taken from [23]

11111111 11111111 11111111) - about 4.3 billion unique IPs - clearly not enough in current world of internet [20]. The solution is simply to use IPv6 (current 32 bit addresses are IPv4), which allows up to 128 bits being stored, or up to about $3.4 * 10^{38}$ (340 decillion) - this would allow to assign a unique IP to each grain of sand on earth, and so it is definitely enough for current (and foreseeable) needs [20]. The problem is that the transition from IPv4 to IPv6 is not easy (ie. all hardware would have to adapt), and so meanwhile our internet service providers assign 1 unique IP address to our router, the gateway, and this router will assign private addresses throughout our private network [20]. This means that all hosts within the network needing to contact outside world, and all the outside world contacting hosts within our network would go through the router.

After understanding why our addresses are so similar and what the gateway IP is, let us have a look at the four lab tasks:

1. SYN Flooding Attack
2. TCP RST Attack on telnet and ssh Connections
3. TCP RST Attack on Video Streaming Application
4. TCP Session Hijacking

We will explore just the first task as they are all about exploiting TCP/IP:

“SYN flood is a form of DoS attack in which attackers send many SYN requests to a victim’s TCP port, but the attackers have no intention to finish the 3-way handshake procedure.” [23]

One must understand TCP protocol in a slightly better detail than I have described in order to properly understand SYN flood attack, but I will try to keep the explanation succinct. As

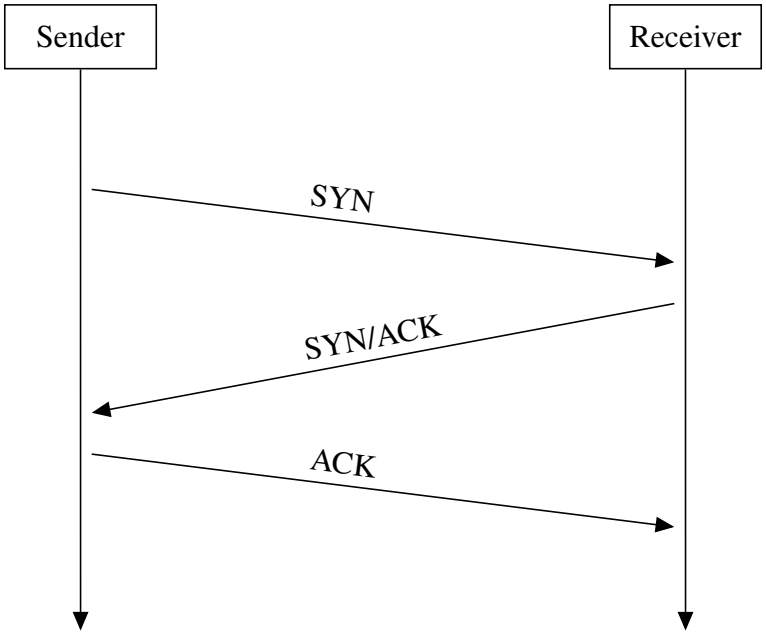


Figure 1.3: Simplified 3-way handshake

mentioned above, TCP is a *connection-oriented* and a *reliable* communications service. This means that there needs to be a way to ensure that the packet we sent was received. There is; the receiver simply must respond after he receives a packet and hence lets us know that the packet was received. However, in order to establish this connection there must be what's called a *3-way handshake*. In 3-way handshake the sender sends a packet saying that he wants to establish a connection (SYN packet), the receiver sends a response that he is ready for connection (SYN/ACK packet) and lastly, the sender receives the server's response and sends ACK packet to establish the connection. [20] This can be observed at (a very simplified) Figure 1.3.

Through this attack, attackers can flood the victim's queue that is used for half-opened connections, i.e. the connections that has finished SYN, SYN-ACK, but has not yet gotten a final ACK back. When this queue is full, the victim cannot establish any more connections and hence we denied service to other users. [23]

Using tools like netwox to conduct the attack (netwox let's us create artificial packets), and then we use a sniffer tool (it *sniffs* traffic to display packets and its details) to capture the attacking packets. Checking the state of our queue before and after sending the artificial packet, we can see that it is rather simple to fill it up and hence disable further connections.

The only real downside to this game was the environment setup. The first task taught one of the most common DoS [35] attacks, which is important to be taught to any computer security student. It did not, however, teach how about some countermeasures of the attack, ie. how to protect us from SYN flood attack - which is equally important to teaching about *what* the attack is and *how* it works. Two examples of countermeasures would include [13]:

- Filtering
- Reducing the SYN-RECEIVED Timer

1.2.1.2 Heartbleed Attack Lab

The Heartbleed bug (CVE-2014-0160) is a severe implementation flaw in the OpenSSL library, which enables attackers to steal data from the memory of the victim server [33]. The contents of the stolen data depend on what is there in the memory of the server. It could potentially contain private keys, TLS session keys, user names, passwords, credit cards, etc. The vulnerability is in the implementation of the Heartbeat protocol, which is used by SSL/TLS to keep the connection alive. [21]

Similar to previous lab, we need multiple machines, this time two (attacker and victim). As you can see there is a pattern in these games - they deal with network security and network by definition is at least two machines.

To understand how heartbleed protocol works, let us look at it at Figure 1.4 (figure from [21]). As you might see, server respond depends on user request. The actual bug was in an insecure malloc (fix by OpenSSL: [17]):

```
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
```

There were no checks performed on the length of the *payload* + *padding* and so user could force the server to respond with arbitrary memory location.

The lab uses `attack.py` [19] attack script originally written by Jared Stafford to exploit the vulnerability of heartbleed protocol and read what is in server's memory. Next it tries to find the cause of the bug, and lastly it tries to fix the bug. Overall an interesting and engaging lab. This time the lab also taught about countermeasure, yet for me the biggest weakness remains - the lab still requires two machines and this might deter potential students from attempting it.

1.2.2 Google's XSS-game

Next let us have a look at a bit different game - it has nothing to do with firewall per se, yet it deals with computer security and the areas overlap.

The XSS game [18] is divided into levels. Once you finish a level, it allows you to progress to the next level (ie. you cannot skip). If you are stuck, it provides hints. Obviously, the game gets harder in each level.

Cross-site scripting (XSS) bugs are one of the most common and dangerous types of vulnerabilities in Web applications. [18] In simple terms, XSS exploits allow us to execute code, where it was not originally intended for us to do so. Let us have a look at the first level (out of 6) found at Figure 1.5.

Moreover, the game let's us see the code behind (to see how the website is implemented); a short snippet (showing the vulnerability too) can be seet at Figure 1.6.

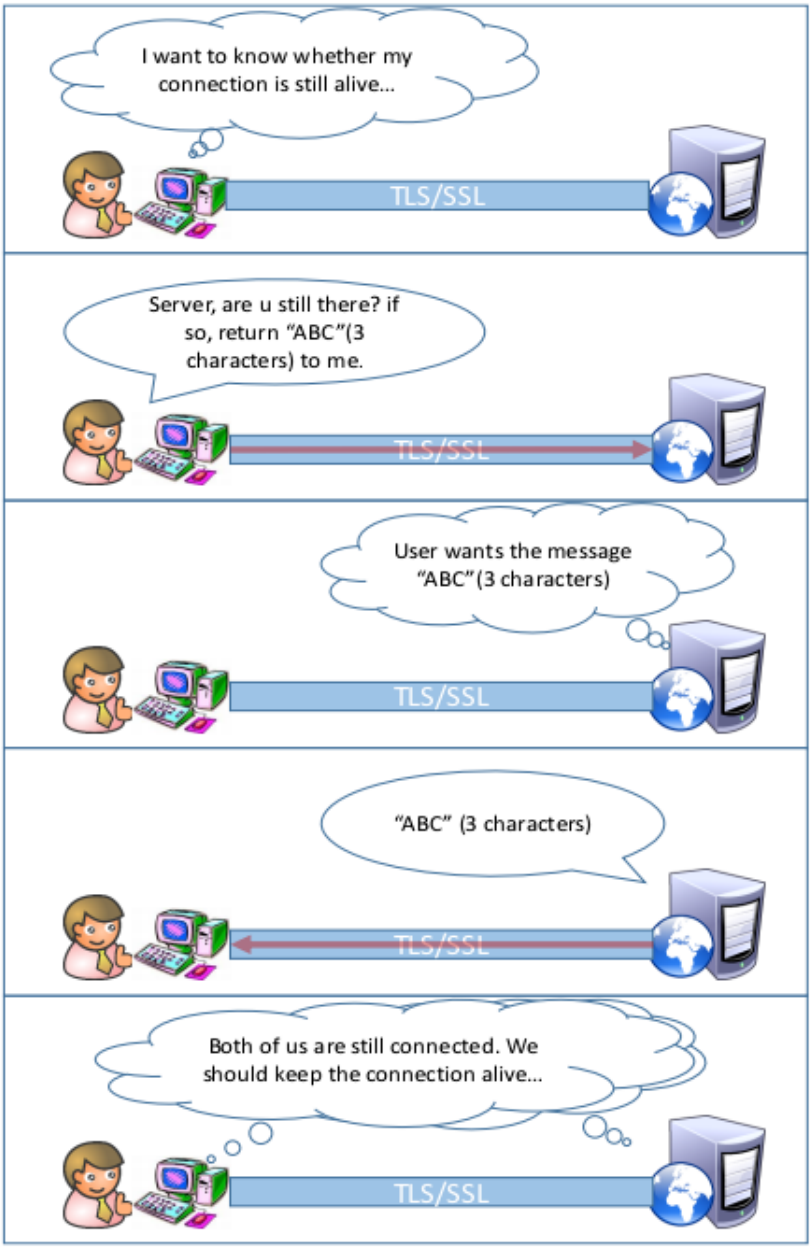


Figure 1.4: Heartbleed protocol

Figure 1.5: Google's XSS-game, level one

```
44        # Our search engine broke, we found no results :-(
45        message = "Sorry, no results were found for <b>" + query + "</b>."
46        message += " <a href='?'>Try again</a>."
```

Figure 1.6: Snippet of code showing the bug

As you might observe, there is no sanitization on the `query` and so any HTML code can be injected. To exploit it we simply put:

```
<script>alert(1)</script>
```

into the query box and hit Search - this executes the JavaScript code and alert message pops up. As expected, a very simple task for first level to get us started.

A very interesting and entertaining game overall. I loved the fact that I could simply go to a website and dive straight into trying to complete the task, instead of spending hours on figuring out how to setup the environment (like in the first two games mentioned). This game game me the initial idea about my project, which is described in the next section (1.3).

1.3 Initial Idea

Overall, the XSS-game was super interesting and I thought I would love to develop something like this - so I went to ask William and Kami about it. They gave me permission to do so and so I went on to look at how to make a website (as I have never done so before), how to program in JavaScript (and HTML, although that is not really programming) - these areas will be covered in later chapters.

Then the reality hit me and I realised that there is no way I could accomplish something as complex as the google's game (ie. code to check other people's code without my code being vulnerable, when the environment I am trying to do it in is new to me). Sad and defeated, I looked back at *why* have I not enjoyed the lab games and realised that it was simply because of the overhead of doing them, not the actual content. Therefore, I decided to see if I could make a simulator where students in similar situations as me could skip the environment requirements and dive straight into doing the tasks instead.

I have already discussed how firewall (section 1.1) and TCP/IP protocol work (subsection 1.1.1), however, there is one thing left to discuss before this chapter is concluded - an actual implementation of firewall: `iptables` [7] - an implementation I was trying to replicate in my WebApp simulator.

1.4 IP tables

In this section I will briefly cover the idea behind IP tables, the IP table rule, adding, deleting and inserting a rule to a table and how the packets are filtered using the rules.

`iptables` is the userspace command line program used to configure the Linux 2.4.x and later packet filtering ruleset. It is targeted towards system administrators. [8]

In essence, `iptables` is a firewall that is accessed through command line by system administrators. It accepts set of commands in order to configure the firewall behaviour.

Note that `iptables` applies to IPv4 and throughout this report there is the assumption that we always refer to IPv4 and not IPv6 when referring to an IP address. `ip6tables` would usually refer to IPv6 packet filter configuration. [15]

1.4.1 IP table chain

`iptables`, as the name suggests, contain `tables`. These tables contain `chains` and these chains contain `rules`. [15] In this report we are concerned with the `filter` table - ie. the table responsible for filtering out packets in our firewall.

Packets are processed by traversing the rules in chains one after another. A rule in a chain can cause a the packet to jump to another chain, and this can be repeated however many times. There are three pre-defined chains in filter table [15]:

- *INPUT* - packets that are inbound to our host enter this chain
- *FORWARD* - packets that are not 'aimed' at our host (ie. just passing through us)
- *OUTPUT* - packets trying to leave our host the hardware

Moreover, the administrator can define his own chains to manage his work better. For instance we could make a chain called `FRIENDS` which will contain the IPs of our friends - but we will cover that later on in examples (subsection 1.4.4).

Each rule consists of specifications which determines what packets it matches. As each packet goes through the chain, the chain examines whether the packet matches the rule, if it does, then it looks at what the rule says should be done with the packet; this is called `target`. [15] If it does not match the rule, next rule is examined until either a rule matching the packet is found, or the list of rules is exhausted.

1.4.2 IP table rule

As already mentioned, the chains within the tables consists of set of rules that determine the fate of packets. An `iptables` (firewall) rule specifies criteria for a packet and a target [15]. If a packet matches with a rule, then we look at what the value of target is, which can be one of three things:

1. A user-defined chain (e.g. `FRIENDS`)
2. One of iptables-extensions (out of scope for this report)
3. One of special values (`ACCEPT`, `DROP` or `RETURN`)

If a user-defined chain is encountered, we simply let that chain handle the packet (ie. the rule will, again, traverse the rules within that user-defined chain until it finds its fate).

Otherwise, if the special value is encountered, the fate is decided - as can be seen at Figure 1.7.

An example of an iptable rule would be:

```
iptables -A FRIENDS -j ACCEPT
```

Value	Fate
ACCEPT	let the packet through the firewall
DROP	do not let it through the firewall
RETURN	stop traversing this chain and return to the previous chain this packet came from

Figure 1.7: Special `target` values and the behaviour

This rule would match every packet that comes to chain `FRIENDS` and let it through the firewall. We will examine more examples in subsection 1.4.4.

1.4.3 Adding, deleting and inserting a rule

Knowing what a `table` is, what a `chain` is and what a `rule` is, the next question is: how do we add, delete and insert them into chains?

The command is as follows:

```
iptables [-t table] {-A | -I | -D} chain <specification>
```

However, as already mentioned, we only deal with `filter` table and omitting the table -t command automatically defaults to `filter` table, hence in all future examples (and implementations) this will be omitted.

Specification refers to the criteria (ie. what packet matches the rule) and the target (ie. what is supposed to be done with the packet once it matches). [15]

1.4.4 Examples

Let us explore some basic `iptables` settings to improve our understanding. All the examples will consist of set of commands which add (insert or delete) rules to filter table.

First example:

```
iptables -A INPUT -s 8.8.8.8 -j DROP
iptables -A INPUT -j ACCEPT
```

We really do not like Google and so whenever a packet arrives from IP 8.8.8.8, we drop it, otherwise everything else is accepted (let through).

Second example:

```
iptables -A INPUT -s 8.8.8.8 -j GOOGLE
iptables -A INPUT -j ACCEPT
iptables -A GOOGLE -j DROP
```

This achieves precisely the same functionality, however, we use a user-defined chain (`GOOGLE`) to handle the fate of the packets.

Third example:

```
iptables -A INPUT -s 8.8.8.8 -j GOOGLE
iptables -A INPUT -j ACCEPT
iptables -A GOOGLE --dport 22 -j DROP
iptables -A GOOGLE --dport 80 -j DROP
iptables -A GOOGLE -j ACCEPT
```

This is where we can show the benefit of having user-defined chains. If the IP is recognised as belonging to `GOOGLE` chain, then in `GOOGLE` chain we do not have to deal with the IP anymore (ie. we know it belongs to us), we just deal with the logic of what to do with it. In this case if the packet tried to request port 22 (ssh) or port 80, it was dropped, otherwise we let it through (`ACCEPT`). The real benefit, however, is when we want to match multiple IPs as `GOOGLE` while also having a bit more complicated logic than simple `DROP` or `ACCEPT` policy:

```
iptables -A INPUT -s 8.8.8.8 -j GOOGLE
iptables -A INPUT -s 8.8.4.4 -j GOOGLE
iptables -A INPUT -j ACCEPT
iptables -A GOOGLE --dport 22 -j DROP
iptables -A GOOGLE --dport 80 -j DROP
iptables -A GOOGLE -j ACCEPT
```

As you see, adding a new IP to the `GOOGLE` chain is really trivial and we keep the logic of what should happen to that packet separated (ie. nothing in `GOOGLE` chain changes; `INPUT` chain just forwards one more IP to `GOOGLE` chain). This let's us be much more scalable while keeping the logic separate from the IPs.

Chapter 2

Design and Implementation

Knowing what we wanted to do - re-implement `iptables` as into a WebApp firewall simulator, the question is what *exactly* should it do, and *how* should it do it. In this chapter we will discuss precisely these details, and explain how does it help the people using it.

2.1 Goals

The original goals of the project were as follows:

1. Make a WebApp with two terminals, `outsider` and `insider`, where, as the names suggest, one would act as a machine outside the network, and one as inside the network

2. Allow adding, inserting and deleting `iptable` rules

3. Allow using `nmap` to 'probe' the other terminal. That is, send a packet to the other terminal and examine the response - is the packet `ACCEPTED` or `DROPPED`?

This would let future students to play around with firewalls, without the need to spend hours on setting up the environment to do so. They would no longer need multiple machines or anything alike. All they would have to do is open a website in their browser and there they would be presented with two terminals, both simulating `iptables` environment.

Albeit useful, I realised that it would not be possible to implement the full functionality of the `iptables` within the timeframe I was given, nor with my knowledge about the domain when I started, and so I outlined some reasonable goals (ie. #2 and #3) that I thought I could achieve within the timeframe allocated.

However, in order to meet these goals, there were several things, requirements, for me to research, learn about and learn to work with before I could achieve the goals.

2.2 Requirements

Some of the things I needed to research were already mentioned in the background chapter - ie. how firewalls work and how `iptables` work. While it took a lot of time to learn about the firewalls and `iptables`, it was mostly just reading a lot of material - manuals, articles, books and code; not necessarily challenging, simply time consuming. The real challenge, however, was that I never made a website before and so I had to learn how to do so from scratch - this included learning HTML, JavaScript, CSS and some frameworks like Node.js (NPM). The next challenge was to implement the `iptables` the way I outlined in goals section (section 2.1), but that will be covered later on (subsection 2.4.2).

2.2.1 HTML, CSS and JavaScript

While HTML [29] was rather straightforward, CSS [25] proved to be a quite challenging - particularly implementing the terminals so they behave like terminals. I am not quite sure whether this was due to me being completely new to CSS, or that it was actually challenging. Making the cursor blink realistically was certainly not easy for me when I started!

JavaScript [26] was a completely new language to me. I was familiar with the languages used in our courseworks - C, Java, Python, Haskell etc; but JavaScript was nothing alike. The project was developed in ECMAScript 6 [14] - which was the latest JavaScript version (at the start of project; October 2016).

While there were aspects I was used to, e.g. Python, like JavaScript, is dynamically types, there were things I was totally mind blown about - e.g. the language uses == and === for comparison. Why?!

```
1 '' == '1'           // false
2 1 == ''             // true
3 1 === ''            // false
4 1 == '1'            // true
5 1 === '1'           // false
6 '1' === '1'         // true
```

Eventually I found out why about this, and many other features that were different (for this one in particular === behaves exactly like == but no type conversion is done and the types must be same in order to be considered equal). I am still learning about the new features of JavaScript, with the recent ES6 [14] and its revamp on OOP and JavaScript.

2.2.2 Node.js, NPM and other frameworks

Node.js [16] is an open-source, cross-platform JavaScript runtime environment for developing both back-end and front-end applications. It basically let's us develop the back-end and the front-end using the same environment, and the same language (JavaScript). Although in this project we only developed front-end application, as it was not deemed necessary to also have a back-end.

The big advantage of using Node.js is that there are open-source packages available on *search.nodejs.org* that were developed by other people and that make development life so much easier, while also making the development cycle much shorter.

To manage these packages (ie. install, version control etc), Node Package Manager (NPM) [27] is used.

I will list some of these packages to stress the importance and how much time they saved.

Gulp.js [24] - this package is truly a life changer for development. Basically it builds your project automatically for you, any time you make a change (if setup to do so!). Let us consider the tasks that are usually required to be done any time a change is made (in my project in particular):

- removing `console` and `debugger` statements from scripts (or commenting them out)
- transpiling ES6 to cross-browser-compatible ES5 code (using **Babelify.js** [1])
- concatenating and minifying CSS and JavaScript files

These need to be done *every time* anything is changed - this is actually accomplish, by yet another, package called **Watchify.js** [6] that looks for changes and triggers the compiling. Manually, these builds take about 30 seconds to do; sure, that's not a lot of time. Now imagine having to do that 100 times a day, that's 3000 seconds or 50 minutes spent on just building the project. Every day. The point is that it gets super tedious and this package does it automatically. It saved me huge amount of time over the course of developing this project. Of course, I had to get familiar with the package and how to use it, which took time on its own, but that time is nowhere near compared to how much time was saved using the package.

Next super useful package - **Browserify.js** [3]. This package allowed me to structure the project into multiple files - ie. in OOP fashion where mostly each class has its own file and so it is easier to navigate and make changes in future. In essence it allows exports of functions/classes in a given file, and importing it using `require` in other files.

QUnit.js [5] was used to run unit tests during the development which is later explained in section 2.5.

Last (important) package that I will mention was **Jshint.js** [4] which I found particularly useful at the start of my development when I was not yet very confident in writing JavaScript code. The problem with writing JavaScript code is that it allows syntax errors to go unnoticed. For instance, it is standard to end your lines with a semicolon, but JavaScript will compile and work even if you do not do so. It can, however, lead to unexpected behaviour. Jshint looks at code and gives warnings to the developer whenever it spots something which is (probably) not intended.

The full list of packages (and dependencies), along with their version that the project used can be seen at Figure 2.1.

Package	Version
babel-preset-es2015	v6.16.0
babelify	v7.3.0
browserify	v13.1.0
file-saver	v1.3.3
gulp	v3.9.1
gulp-jshint	v2.0.1
gulp-sourcemaps	v2.1.1
jquery	v3.1.1
jshint	v2.9.3
jshint-stylish	v2.2.1
underscore	v1.8.3
vinyl-buffer	v1.0.0
vinyl-source-stream	v1.1.0
watchify	v3.7.0

Figure 2.1: Project dependencies

2.3 User Interaction

In this section I show how the WebApp looks like and how is user supposed to interact with it (Figure 2.3.1); and what commands are supported (subsection 2.3.2).

2.3.1 Graphical User Interface

My intention at the very beginning was to keep the graphical user interface (GUI) as simple as possible. Originally, it would have just been two terminals (`outsider`, `insider`). However, upon suggestions of my supervisor and Kami Vaniea I have also added buttons which allow users to upload and download rules in bulk (so multiple rules can be edited easier). The buttons use simple `Bootstrap` [2] font to make them look better than plain HTML.

The terminals (`outsider`, `insider`) along with the upload and download buttons can be seen at Figure 2.2.

I tried to make the terminals behaviour as similar to a regular linux terminal as possible. Specifically, if arrow-up is pressed, the previous message typed is displayed (same for arrow-down). Similarly, when a command is not recognised, the expected `bash: <command> not found` is as seen at Figure 2.3. The background color was chosen to be black, as the website background is white (and so white terminal with black font would not be very visible).

The user interaction is (hopefully) very simple - users are expected to type in the commands. If the command is not supported, system tells them so, if it is supported, the same behaviour as in linux terminal is expected to happen in the WebApp terminal.

Figure 2.2: `outsider` and `insider` terminals as presented to the user

Figure 2.3: Command not found

2.3.2 Supported commands

Figure 2.4 shows the currently implemented commands and explaining their behaviour.

2.4 System Structure and Implementation

As previously mentioned, the simulator is developed as a WebApp using just JavaScript, HTML and CSS, however, I have decided to make the WebApp as a static webpage in a sense that there is no back-end server needed to evaluate the queries. Everything is done locally, in user's browser and saved in the local storage of the browser. This has two main advantages:

- There is no response time to user's queries and so the user experience is much smoother
- It keeps the project structure and deployment much simpler, as there is just front-end to take care of

There are some disadvantages though, for instance:

- There is no possibility of creating user accounts and storing their firewall settings for them
- User can temper with code of the website in the front-end, making the behaviour of firewall deviate from what was intended. This, however, I do not consider as a problem

Command	Behaviour
iptables ...	`iptables` commands. These will be explained in greater detail in functionality subsection (subsection 2.4.2 at Figure 2.6)
nmap -p port ip	tells us whether a port at certain ip accepts or drops packets
set ip new_ip	custom command that sets ip of the terminal it was typed in to a new_ip
clear	clears up the messages in terminal. It is, however, still possible to see navigate to previous messages using up and down arrows
ipconfig	displays current ip address of the terminal (functionality not identical to real environment)

Figure 2.4: Supported commands and their behaviour

because this is an educational tool in the end. It would be of no benefit for the user do this

2.4.1 Files and Classes

The file structure of the project can be found at Figure 2.5.

`README.md` contains instruction on how to run the system.

`LICENSE` contains MIT license.

`package.json` contains the dependencies, ie. what packages and their versions does Node.js need.

`gulpfile.js` file (in `firewall` not `testsuit`) contains the script which compiles everything as explained in subsection 2.2.2. The `gulpfile.js` in testsuit compiles the separate tests into one big test file (`main_test.js`) and the results (ie. what passed/failed) can be seen in `result.html`.

`util.js` contains some utility functions and values (e.g. keycodes for key presses).

Once everything is compiled, there are just 3 files needed to load the whole website:

1. index.html
2. styles/style.css
3. dist/build.js (circa 1500 LoC)

The `build.js` file, however, is compiled from a number of different JavaScript files containing the logic of the website. These files are contained within the `src` folder.

The `app.js` file ties everything together. It creates the two Terminal objects (outsider and insider) and when user leaves the website, it saves the messages written, the IP of the terminals

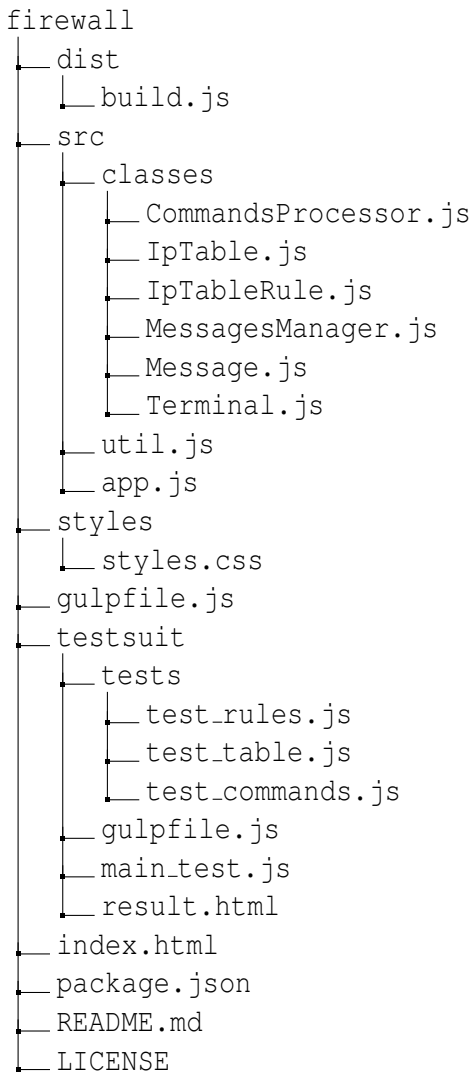


Figure 2.5: Project file structure

and the firewall rules so that in future user does not have to set these properties again. It also initialises CommandsProcessor object which is passed to both Terminals and it is used to evaluate user input.

The terminal object creates its own MessageManager and IpTable objects.

`MessageManager` is responsible for storing everything user has typed and loading it up when user visits the website again. It also has the logic of key presses, ie. arrow-up, arrow-down and return keys.

`IpTable` is responsible for storing IpTableRule objects using various methods (e.g. add, insert, delete). It can also list the rules, and determine whether a certain port is opened. This is where the main logic of the firewall is executed.

`IpTableRule` is an object that represents iptable rule within the iptable. It is constructed using builder pattern [34]. The reason is that it has too many attributes to be passed into a constructor method, yet a lot of these attributes can be omitted to default to certain value (e.g. if no `source` is specified, the rule `anywhere` as source address). Hence builder pattern is used and in JavaScript I implemented it as function called `rule()`:

```
1  var rule = function rule() {
2      return {
3          info: { //default values
4              // some values
5              source: "anywhere",
6              // more values
7          },
8          // some setters
9          source: function source(src) {
10             this.info.source = src;
11             return this;
12         },
13         // more setters
14         build: function build() {
15         // constructs the IpTableRule object and returns it
16             return new IpTableRule(this.info);
17         }
18 }
```

In practise, this works really nicely. An example of how an IpTableRule is constructed is shown below:

```
var newRule = rule();
newRule.protocol('UDP');
newRule.source('192.168.0.1');
newRule.jump('ACCEPT');
ipTable.addRule(newRule.build());
```

Admittedly, the methods should have been called `setProtocol(p)`, `setSource(s)` and so on, however, there are no real getters and setters in JavaScript (unlike Java) and these methods are only used when constructing a rule using the builder, hence I saw no need to call them setters.

`CommandsProcessor` is where the user input is processed. It stores the logic of 'is this command recognised? If so, do this and this, if not, respond like this'. It is here where I had to build a small parser in order to parse the rules, and while my implementation has an obvious flaw (discussed in section 2.6), I worked under assumption that the user has no ulterior motives but to save time setting up the environment and use this as an education tool.

2.4.2 Functionality

Some of the basic functionality was already mentioned above when I was describing the classes and files, yet the core functionalities of the simulator are explained below.

Let us have a look at how IpTableRule processing works:

- The first condition is that it must start with `iptables` **not** followed by `-L`. If user calls `iptables -L` then the chains and their respective rules are simply displayed.
- At this point we know that the user tries to:
  - Delete all chains (and hence all rules)
  - Delete a certain chain
  - Delete a rule from a certain chain
  - Create a rule and either append it or insert it

  What I did was for each of the possible (supported) options/specifications that IpTableRule can potentially have, I constructed a regular expression that would find it. For instance we know that the jump target would be an alphanumerical string of form:

```
iptables <some parameters> -j chain
```

  And so we know that it starts with some `flag` '-j', followed by some string that matches the regular expression (in JavaScript it would be '\\w+'). Hence what I did is make a function, 'findToken(msg, flag, regexprs)' that takes three arguments:

  - original full message string typed in the by the user
  - flag (ie. '-j' for the example above)
  - a JS object containing the regular expressions in a key:value pairs. This is because all attributes have a single flag that starts it, but some have multiple values followed - for example `-D chain position` would delete a rule from `chain` at certain `position`. An example of this argument would be:

```
{ chain: "\\w+", position: "[0-9+]" }
```

    This function would return the regexprs object it was given, but the values are replaced by the values found within the message. Additionally, it also adds a new key:value pair, `found` which is a boolean stating whether the information was found in the message. For instance if this message was supplied:

```
iptables -D FRIENDS 4
```

    And the function `findToken` would be called with `flag` '-D' and the `regexprs` as mentioned above, it would return:

```
{ found: true, chain: "FRIENDS", position: "4" }
```

    For our initial example with jump, let us consider this input:

```
iptables -A INPUT -j ACCEPT
```

This would be passed to `findToken`:

```
findToken("iptables -A INPUT -j ACCEPT", "-j", { chain: "\\w+" })
```

And this would be returned by `findToken`:

```
{ found: true, chain: "ACCEPT" }
```

As you can see, using this methodology, I made it rather modular and easy to extend to accommodate new options. The `delete` and `jump` options implementation is illustrated below:

```
1 const d      = this.findToken(msg, "-D",       { chain: "\\w+", position: "[0-9+]"});
2 const j      = this.findToken(msg, "-j",       { jump: "\\w+" }                     );
```

Being able to parse the rule, we now need to build up the `newRule` as each of these tokens correspond to some logic or an attribute of the rule. This time, only `jump` will be used as an example because `delete` refers to action that needs to be taken, rather than setting up an attribute.

```
1 if (j.found)
2     newRule.jump(j.jump);
```

Once the rule it built up, we must decide what action do we want to carry out (ie. one of the 4 options mentioned above - (1) delete all chains, (2) delete a certain chain, (3) delete a rule from a chain, or (4) create a rule and either append it or insert it). `delete` will be used as an example:

```
1 // other conditions
2 else if (d.found)
3   this.terminal.ipTable.deleteRule(d.chain, d.position);
4 // other conditions
5 else
6  this.error(); // invalid action
```

Obviously, this is a simplified example to show parsing of the rule, how attribute is set and how an action is carried out. Source code is available to view the exact details.

All the possible options of `iptables` rule in my implementation are shown at Figure 2.6.

These flags are an adjusted subset of all the flags and options of `iptables`. [15] Not all of the functionality is supported and some of the functionality is not identical compared to the current `iptables` (e.g. my implementation does not support accepting anything other than IP address as `source` and `destination`, whereas the original implementation of `iptables` would also accept a hostname or a network name). That being said, everything developed or tested on using my implementation would be also valid on the original `iptables` implementation it was based on. I have tested it and discuss it in section 2.5.

Flag	Usage
`-A`	**append** *chain rule-specification* Append a rule to the end of the selected chain.
`-I`	**insert** *chain rulenum rule-specification* Insert a rule in the selected chain at the rule number position.
`-D`	**delete** *chain rulenum* Delete a rule at rule number position from the chain.
`-F`	**flush** *[chain]* Flush the selected chain. If no chain was selected, then all chains are flushed. This is equivalent to deleting rules one by one.
`-s`	**source** *address[/mask]* Source specification. Address is a network IP address (ie. with /mask) or plain IP address. Currently only IPv4 addresses are supported.
`-p`	**protocol** *protocol* The protocol of the rule or of the packet to check. The specified protocol can be one of tcp, udp, or the special keyword ”all”.
`-j`	**jump** *target* This specifies the target of the rule, ie. what to do if packet matches it. The target can be user defined chain or one of the special built-in targets which decide the fate of the packet immediately (e.g. `ACCEPT`).
`--dport`	**destination-port** *port* Specifies the port of the rule or of the packet to check. A number from 1 to 65535 is valid.

Figure 2.6: iptables rule options

nmap

The last (important) functionality of the firewall is *probing* the other terminals and seeing whether the packet we sent is accepted or dropped. This can be done using `nmap` [30] and in our implementation it would look like, for example:

```
nmap -p 22 123.123.123.123
```

The response would be:

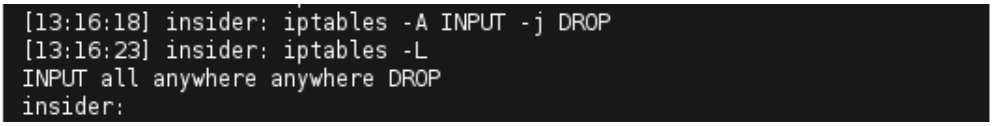
This is because my simulator will strictly communicate just between the two terminals (outsider, insider) and not to the outside world. Ip 123.123.123.123 does not belong to either of them and hence it is an invalid request. What we can do though, is check our IP using `ipconfig` command, ie:

There are two things to note about this:

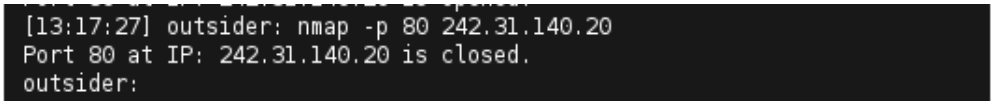
- The IP initially assigned is random
- All ipconfig does is print out our IP. This is not standard behaviour, however, it is enough for our project

Now if tried to `nmap` using the actual address, the response would be:

This is because we have not setup our firewall rules yet. Were we to make a rule that drops all packets coming to the insider terminal:



And execute the same `nmap` command from the outsider, the response would be:



This being said, we could simply call `nmap -p 80 insider` in order to achieve the same result. Internally the terminals can reference each other by `insider` and `outsider`.

Obviously, one could get more creative with the ruleset. Let us take the third example from the background chapter 1.4.4:

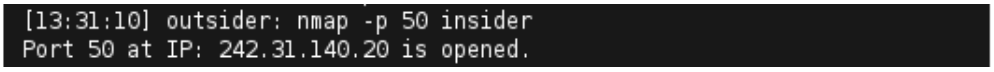
```
iptables -A INPUT -s 8.8.8.8 -j GOOGLE
iptables -A INPUT -j ACCEPT
iptables -A GOOGLE --dport 22 -j DROP
iptables -A GOOGLE --dport 80 -j DROP
iptables -A GOOGLE -j ACCEPT
```

Instead of typing these rules in one by one (which at this point would still be ok, but if we had 100 rules, that might get tedious), we can simply make a file `ourrules.rules`, insert these rules there, and upload it to the `insider` terminal on the website (using the button there).

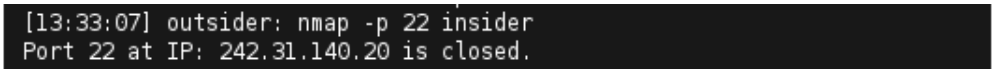
Next, we can pretend to be Google and set our ip in outsider to 8.8.8.8 by simply calling `set ip 8.8.8.8`.

Let us try three commands:

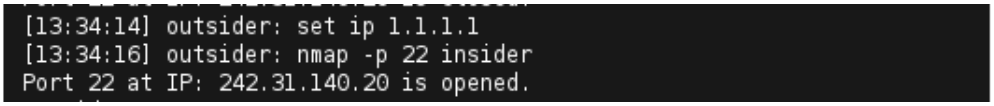
1. A packet which we send from Google's ip, but is not 22 or 80 and hence we expect to be accepted.



2. A packet which we send from Google's ip, but is 22. This should be dropped.



3. Lastly, a packet from IP other than Google - it should be accepted.



This all behaves as expected and the implementation matches the goals we have set for this project. But how do these statements get evaluated? I implemented the procedure explained in the background chapter, where as the packet arrives to the firewall, the rules within the firewall gets examined. First, we look at whether there is some rule in `INPUT`. Each rule is checked whether its specification matches the packet, and if it does, then it determines the fate of the packet (e.g. forward it to other chain, accept it, drop it, return it etc).

Please note that throughout these examples we were using two terminals (outsider, insider), however, because they both have black background I found it extremely wasteful to keep inserting screenshots of the whole webpage and so I just inserted snippets.

Downloading and Uploading rules

User has the ability to download the ruleset he made, and upload a new ruleset. This logic is handled in the Terminal class and was added just recently after it was suggested (during the last stage of testing) that it is quite time consuming to re-type the rule set every time (instead of being able to edit it in plain text file and then upload it to the website). Each

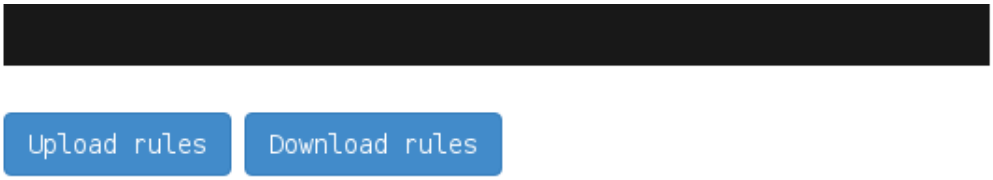


Figure 2.7: Button of terminal along with its buttons under it

terminal has two buttons - `Upload rules` and `Download rules`, which are located just under it as can be seen at Figure 2.7.

Netmask

Last functionality that I will mention and conclude this section is that ip addresses also support network masks. That is, a rule can be specified to block all packets coming from ip address that starts with `8.8`:

```
iptables -A INPUT -s 8.8.0.0/16 -j DROP
```

The `/16` part specifies that we only really care about the first 16 bits, ie. first two out of four bytes of the address.

The behaviour can be showcased below. First, we set the rule (in `insider`):

```
[14:33:17] insider: iptables -A INPUT -s 8.8.0.0/16 -j DROP
[14:33:19] insider: iptables -L
INPUT all 8.8.0.0/16 anywhere DROP
insider:
```

Then we try probe it from `outsider` using `nmap`:

```
[14:33:34] outsider: ipconfig
Your ip: 8.1.1.1
[14:33:42] outsider: nmap -p 22 insider
Port 22 at IP: 242.31.140.20 is opened.
[14:33:48] outsider: set ip 8.8.8.1
[14:33:51] outsider: nmap -p 22 insider
Port 22 at IP: 242.31.140.20 is closed.
outsider:
```

As you can see, initially, with ip address `8.1.1.1` the packet was not dropped. As soon as we changed our ip to `8.8.8.1` and tried to `nmap` again, the packet was dropped.

2.5 Testing

There were two ways I have tested my implementation.

- Construction of the IpTableRule using the builder pattern, inserting or adding it into a table, deleting it from a table and all of the similar functionalities were tested using a node.js framework QUnit.js [5] after anything was changed within the project, to ensure that the core functionality was not affected and was behaving as expected.
- I used VirtualBox [28] to run two virtual machines of linux and tested whether the behaviour of my implementation matches the expected behaviour in the original `iptables` implementation.

This iterative method, ie. implement new feature -> test core functionality using local QUnit tests -> test if behaviour matches with the virtual machines, ensured that while my implementation is not a perfect copy of `iptables`, the features that I have implemented behave identical.

2.6 Discussion

As I mentioned earlier (2.4.1), my implementation is not perfect and can be abused to create rules which are not valid. For instance this rule would be valid in my simulator, but would be not valid in real `iptables` implementation:

```
iptables –D INPUT 4 –A INPUT –j ACCEPT
```

This command does not make sense. It first requests to drop rule 4 in INPUT chain, but then in the same command to append a rule into INPUT chain. However, my implementation would allow such rule (and in current version it would just append the rule without deleting). This was the tradeoff of making my implementation really modular and simple to read/extend, yet not perfect.

That being said, it can definitely be fixed by implementing some extensive function that will validate the rule, returning boolean saying whether the rule is valid (and makes sense) or not. This was out of scope for my project though.

All this being said, I believe I met the goals (section 2.1) I have set.

Chapter 3

Evaluation

This chapter will outline the findings of what users and experts think about usability of this project, and whether it is suitable for educational purposes.

Two forms of evaluates were carried out; (1) survey using System Usability Scale on the project presentation day, and (2) discussion with experts whose area of expertise overlaps firewalls to see what they think about it.

3.1 Survey using System Usability Scale

System usability scale [11] was developed when the industry felt a need to find a tool with which we can evaluate the usability of the systems we produce. It is not easy to evaluate a wide variety of systems using the same survey, as the context of what the systems are trying to achieve is different.

“A measure of effectiveness of a word processing system might be the number of letters written, and whether the letters produced are free of spelling mistakes. If the system supports the task of controlling an industrial process producing chemicals, on the other hand, the measures of task completion and quality are obviously going to reflect that process.” [11]

Hence SUS was developed. It contains 10 items using a likert scale . It is not just 10 random items, but rather a deliberate selection from a pool of 50 items in total. The way these 10 were evaluates and ended up on SUS is rather interesting - the team [11] took two systems:

- linguistic tool aimed at end users
- tool for systems programmers

where one is agreed to be really easy to use and the other almost impossible to use. Then people with wide variety of jobs were selected and they rated both systems against the 50 questionson a 5 point likert scale ranging from “strongly agree” to “strongly disagree”. The items leading to the most extreme responses from the original pool were then selected. [11]

The survey I used is shown at Appendix A.

Item #	Score	Adjustement	Result
1	5	(5 - 1)	4
2	4	(5 - 4)	1
3	2	(2 - 1)	1
4	3	(5 - 3)	2
5	4	(4 - 1)	3
6	5	(5 - 5)	0
7	1	(1 - 1)	0
8	1	(5 - 1)	4
9	2	(2 - 1)	1
10	5	(5 - 5)	0
		total	**16**

Figure 3.1: SUS score calculation example

3.1.1 Score calculation and example

Scoring SUS is not the usual way a likert scale survey would be scored (ie. add up the individual scores and compare with other participants), but rather there is a specific formula to get the SUS score:

- for odd items (1,3,5,7,9) subtract 1 from the score
- for even items (2,4,6,8,10) it is 5 minus the score
- sum up the resulting scores and multiply by 2.5 to get overall SUS score (range 0 to 100)

It might be easier to understand after seeing the Figure 3.1.

Now we multiply the total (16) by 2.5 to get $16 * 2.5 = 40$.

3.1.2 Participants and their scores

We had 17 participants trying our system and it was explained to each of the participants what a firewall is, what `iptables` are and some examples were outlined (similar to the examples in subsection 1.4.4). Then they were left to to explore the simulator on their own and fill out the survey above.

Their SUS scores, along with their gender and year are shown at Figure 3.2.

3.1.3 Interpreting the scores

First, we can look at the overall statistic (ignoring gender and year) which can be seen at Figure 3.3.

The (mean) average of 70.6 does not tell us much on its own, however, the industry average after conducting this survey 3500 times is about 70 [9], and so this puts our system slightly above the

Participant #	SUS score	Gender	Year
1	62.5	Male	UG3
2	70	Male	UG3
3	55	Female	UG3
4	40	Female	Other (Staff)
5	95	Male	Other (Staff)
6	87.5	Female	UG2
7	77.5	Male	UG4
8	60	Male	UG3
9	72.5	Male	UG3
10	65	Female	Other (Staff)
11	92.5	Male	UG4
12	30	Male	Other (Staff)
13	57.5	Female	UG3
14	67.5	Male	UG2
15	80	Female	UG4
16	95	Female	UG3
17	92.5	Male	UG2

Figure 3.2: Participants, their SUS scores, Genders and Years

Statistic	All participants
count	17
average	70.6
min	30
max	95

Figure 3.3: Basic overall statistic drawn from participants' scores

Statistic	Male only	Female only
count	10	7
average	72	68.6
min	30	40
max	95	95

Figure 3.4: Male and Female disparity statistic

Statistic	UG2	UG3	UG4	Staff
count	3	7	3	4
average	82.5	67.5	83.3	57.5
min	67.5	55	77.5	30
max	92.5	95	92.5	95

Figure 3.5: Year disparity statistic

average. However, it would be also interesting to look at the male and female disparity which can be found at Figure 3.4.

The averages of 72 and 68.6 are not extreme, yet it shows that males think the system is more 'usable' than female.

Finally, probably the most interesting statistic of all is looking at year disparity. This simulator is supposed to be an educational tool and computer security courses could potentially use it. This statistic can be seen at Figure 3.5.

There are few things to note (using interpretation of [9])

- UG2 and UG3 students think our system is *Good-to-Excellent* (82.5 and 83.3 averages)
- UG3 students think our system is *OK-to-Good* (67.5 average)
- Staff people think our system is *OK-to-Good* (57.5 average)
- Moreover, staff have the widest range of scores (30 to 95) and so there are individuals who believe the system is *Best Imaginable* within staff (score 95), while also staff who think it's *Awful-to-Poor* (score 30)

3.2 Expert Feedback

These interviews were conducted after I had conducted my System usability Score surveys. It is important to see what educators this about this project, they are the ones potentially using it in their tutorials, labs or courseworks.

Both of the experts are from University of Edinburgh and a short, 30-45 minutes, discussion was conducted with them. I explained them the domain (firewalls/iptables), the project (WebApp simulator of a firewall), and the potential use (education tool for students). Then I showed them the tool's functionality and recorded what they thought about it.

Expert 1

This expert's background is expanding network of internet cables in Scotland highlands and so he has first hand experience with firewalls. Currently teaches at the University of Edinburgh. He thought that all the goals that I have initially outlined were met (and exceeded in some cases) and he thought that the tool can definitely be used as an educational tool at UG3 level, as long as its accompanied by carefully crafted tutorial, because of its limited functionality.

He noted, however, that it could use some better visual representation in order to explain what is going on in the firewall when packets arrive (or try to leave) the system. That is, for instance show how each rule and chains are evaluated.

Expert 2

This expert's background is cryptography and currently lecturing a computer security course at University of Edinburgh. She thought the system was very well implemented, however, the user-interface was lacking in a sense that user was presented with two terminals and that's all - he has no idea what to do next. I admitted that is true, however, this is supposed to be a tool used in conjunction with a tutorial or lab developed by other people, not a tutorial in itself. It could definitely use some kind of simple default tutorial though, and so this would part of the future suggested work. Moreover, a `man iptables`, and `man nmap` commands suggested by her is also a great idea to be implemented in future.

Otherwise, she asked why have I re-implemented the `iptables` from scratch instead of using the open source version; but I have explained that the whole website is front-end only and so everything is done in browser - the `iptables` implementation is certainly not coded in JavaScript. It is coded in C [7]. Hence I had to re-implement the functionality.

3.3 Discussion

The survey carried out gave us an insight on what users think about the usability of our system. The problem is that there were too few participants to draw any reasonable conclusion - especially when they were further split into sub-groups to analyse the specific gender and year disparity. For instance, although the average score of staff was very low (57.5), it also had the biggest range of scores (30, 95), however, there were only 4 staff that took this survey which means this particular statistic is very unrepresentative. It would be the same case for UG2 and UG4, which had 3 participants each, and similar to UG3 which has slightly more - 7 participants.

Having 17 participants made it very difficult to draw concrete conclusions, however, the overall average of 70.6 could be definitely considered more accurate as the individual group scores, simply because more participants were considered.

What experts thought gave me a great insight into what could be developed in future to extend our project (e.g. better visual representation of what happens to the packets), however, it also

confirmed that I have done something that can be, potentially, very useful to other students.

Next, it must be said that the two seed labs discussed in subsection 1.2.1 that this project was based upon are not possible to actually carry on in my implementation. This has two reasons:

1. The original goal was not to fit my project into the specific environment required in the seed labs. Merely, I just found the seed labs lacking in this area and thought I could fill the gap for other, future, projects of similar nature. That is, were other labs/tutorials developed in future, the instructors can point to my simulator and use it as the environment to carry it out. The project is also open source and so the instructor can adjust it to his needs.
2. Time wise it was not plausible to implement the full environment required by the labs. I only had around 300 hours to spend on practical side this project, of which most was spent on actually learning about `iptables` and firewalls, about website development, usage of the frameworks and then finally developing the project. I tried my best though, to make the project as modular as possible so future extensions and functionality can be added with ease.

Chapter 4

Conclusion and Future work

This dissertation was about firewall game. It started as a simple journey of picking a project that interested me and for me that had to meet two criteria: (1) be a game or an app, (2) be in an area that already interests me. This project matched both criteria and so I selected it. Luckily, I was one of the few people who were given the opportunity to get this project.

Next, I had to read a lot about firewalls in order to understand the area better; up to this point I had maybe one or two lectures worth of experience with the domain - certainly not enough! Then I had to look at what games already exists out there related to firewalls (and security in general). This is where I stumbled upon the idea of my project - making the simulator. It was just too much pain to setup the environment to carry out these games, tasks and labs so I decided to simplify it for other students in future.

Then came spending several weeks on trying my best to both learn more about the domain, learn about making websites, learn about the frameworks I was using, and finally implementing the simulator.

Lastly, I had to find out and evaluate whether what I have created actually makes sense and is useful to other people - this was the trickiest part, but with the expert interviews and insights, I gained confidence that it, indeed, is useful and makes sense.

This chapter concludes this dissertation and the last section will discuss the potential future work that can be carried out to make the project even better and more useful to other people.

4.1 Future Work

Let me start by saying that this simulator is a very simplified version of `iptables`. The most obvious extension that could be made in future is to make it a full version of `iptables`, to support the full functionality.

Furthermore, as Expert 2 suggested in section 3.2, it should really be more user friendly - namely, the man pages should be included, and maybe even add a small tutorial to guide the users through some simple tasks to show them how to use the `iptables`.

Next suggestion would be to make terminal creation more dynamic - ie. many tasks require more than two machines and so let user decide how many terminals he is provided with. This should be rather trivial to implement with the way the source code is setup, it just was not one of my goals and neither was it suggested at any point by my supervisor or other people.

Last suggestion would be to make a back-end server for user accounts storage. I opted out of this simply because I did not have enough experience with building the websites and the front-end solution suited my goals just fine. However, it would certainly be more convenient for users to create accounts and keep their `iptables` configuration stored at the server, that is, if they login from different machine, their `iptables` would simply download from the back-end, instead of being stored in the browser in local storage.

Bibliography

[1] babelify.js. https://babeljs.io/. Accessed: 27-03-2017.

[2] Bootstrap. http://getbootstrap.com/. Accessed: 05-04-2017.

[3] browserify.js. http://browserify.org/. Accessed: 27-03-2017.

[4] jshint.js. http://jshint.com/. Accessed: 27-03-2017.

[5] Qunit: A javascript unit testing framework. https://qunitjs.com/. Accessed: 30-03-2017.

[6] watchify.js. https://github.com/substack/watchify. Accessed: 27-03-2017.

[7] Pablo Neira Ayuso and netfilter. http://git.netfilter.org/iptables/. Accessed: 25-03-2017.

[8] Pablo Neira Ayuso and netfilter core team. The netfilter.org "iptables" project. https://www.netfilter.org/projects/iptables/index.html. Accessed: 25-03-2017.

[9] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual sus scores mean: adding an adjective rating scale. *Journal of Usability Studies*, 2009.

[10] Braden and Robert. Requirements for internet hosts-communication layers. 1989.

[11] John Brooke. System usability scale. https://hell.meiert.org/core/pdf/sus.pdf, 1986.

[12] Wenliang Du. Seed labs. http://www.cis.syr.edu/~wedu/seed/labs.html. Accessed: 01-03-2017.

[13] Wesley M Eddy. Tcp syn flooding attacks and common mitigations. 2007.

[14] Ralf S. Engelschall. Ecmascript 6. https://github.com/rse/es6-features/blob/gh-pages/features.txt. Accessed: 15-10-2016.

[15] Herve Eychenne. iptables man. http://ipset.netfilter.org/iptables.man.html. Accessed: 25-03-2017.

[16] Node.js Foundation. node.js. https://nodejs.org/en/. Accessed: 03-03-2017.

[17] Stephen Henson. Heartbleed b. https://github.com/openssl/openssl/commit/96db9023b881d7cd9f379b0c154650d6c108e9a3#diff-2. Accessed: 20-03-2017.

[18] Google Inc. Xss-game. https://xss-game.appspot.com/. Accessed: 05-03-2017.

[19] Stafford Jared and Seed Labs. attack.py. http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/Heartbleed/attack.py. Accessed: 01-03-2017.

[20] Kurose and James F. *Computer networking: A top-down approach featuring the internet, 3/E*. Pearson Education India, 2005.

[21] Seed Labs. Heartbleed attack lab. http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/Heartbleed/Heartbleed.pdf. Accessed: 01-03-2017.

[22] Seed Labs. Network security labs. http://www.cis.syr.edu/~wedu/seed/network_security.html. Accessed: 01-03-2017.

[23] Seed Labs. Tcp/ip attack lab. http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Networking/TCPIP/TCPIP.pdf. Accessed: 01-03-2017.

[24] Jed Mao, Maximilian Schmitt, Tomasz Stryjewski, Cary Landholt, and William Lubelski. *Developing a gulp edge: the streaming build system*. O'Reilly, Sebastopol, CA, 2014.

[25] Mozilla Developer Network. Css developer guide. https://developer.mozilla.org/en-US/docs/Learn/CSS. Accessed: 01-03-2017.

[26] Mozilla Developer Network. Javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript. Accessed: 01-03-2017.

[27] npm Inc. Node package manager. https://www.npmjs.com/. Accessed: 27-03-2017.

[28] Oracle. Oracle vm virtualbox. https://www.virtualbox.org/. Accessed: 30-03-2017.

[29] Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

[30] seclists.org. Nmap 7.40 holiday release: a dozen new nse scripts, hundreds of new fingerprints, new npcap, faster brute forcing, and more... http://seclists.org/nmap-announce/2016/7. Accessed: 17-01-2017.

[31] Microsoft TechNet. Firewalls. https://technet.microsoft.com/en-gb/library/cc700820.aspx. Accessed: 01-03-2017.

[32] Microsoft TechNet. Tcp/ip protocol architecture. https://technet.microsoft.com/en-gb/library/cc958821.aspx. Accessed: 01-03-2017.

[33] CVE Common Vulnerabilities and Exposures. Cve-2014-0160. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Accessed: 28-03-2017.

[34] Wikipedia. Builder pattern. https://en.wikipedia.org/wiki/Builder_pattern. Accessed: 03-03-2017.

[35] Wikipedia. Syn flood. https://en.wikipedia.org/wiki/SYN_flood. Accessed: 03-03-2017.

Appendix A

System Usability Scale survey

Feedback Survey

		Strongly disagree				Strongly agree
1.	I think that the target audience would like to use this system	1	2	3	4	5
2.	I found the system unnecessarily complex	1	2	3	4	5
3.	I thought the system was easy to use	1	2	3	4	5
4.	I think that I would need the support of a technical person to be able to use this system	1	2	3	4	5
5.	I found the various functions in this system were well integrated	1	2	3	4	5
6.	I thought there was too much inconsistency in this system	1	2	3	4	5
7.	I would imagine that most people would learn to use this system very quickly	1	2	3	4	5
8.	I found the system very cumbersome to use	1	2	3	4	5
9.	I felt very confident using the system	1	2	3	4	5
10.	I needed to learn a lot of things before I could get going with this system	1	2	3	4	5

What is your gender?

☐ Male
☐ Female
☐ Prefer not to answer
☐ Other .....

What year are you?

☐ UG1
☐ UG2
☐ UG3
☐ UG4 / MInf
☐ MSc
☐ Other .....