Implementing a L3, L4 Stateful Firewall with P4

Work Assignment 1 - Software Defined Networks

April 15, 2024

Guilherme Sampaio	**Miguel Gomes**	**Rodrigo Pereira**
PG53851	PG54153	PG54198

1. Introduction

In this work assignment, we implement a L3, L4 stateful firewall using the `P4` programming language (firewall logic and the routing mechanisms) for a fictitious company named `TechSecure Inc.`, with a specific network scenario, built with `mininet`, and a specific set of security rules.

In this report, Section 2, describes the network scenario utilized throughout this project, as well as the specific traffic rules for each local area network, Section 3 exhibits how compile, configure and execute the project and Section 4 handles the specifics of how the routing and forwarding work within the P4 layer to achieve a working firewall. Finally, Section 5 describes the executed tests along with the proposed rules.

2. Network Scenario

The scenario involves a network with three `LAN`'s: **Sales** (`LAN1`), **Research & Development (R&D)** (`LAN2`) and **Management (CRM)** (`LAN3`). Each `LAN` has its own set of servers and hosts connected to a switch, which in turn are connected to the `BMv2 P4` routers. The security rules for the firewall, for each `LAN`, go as follows:

Sales *- LAN 1*

- Access to `CRM` systems via `TCP` on port 443
- Access to `CRM` email servers via `TCP` on ports 25 and 587
- Access to `R&D` Research Server 1 (svr21) via `TCP` on port 80
- Access to `CRM` Financial Data Server 1 (svr31) via `TCP` on port 8080

Research & Development *- LAN 2*

- Access to dedicated research servers on IP's that match `10.0.2.0/24`
- Access to `Sales` email server via `TCP` on port 25
- Access to `CRM` Financial Data Server 2 (svr32) via `TCP` on port 443

Management *- LAN 3*

- Access to financial data servers via `TCP` on port 8080
- Access to financial administrative systems via `TCP` on ports 8080 and 22 (SSH)
- Access to `CRM` system via `TCP` on port 443
- Access to `R&D` Research Server 2 (svr22) via `TCP` on port 22

2.1. Network Topology

In order to fully understand the rule-set defined above, we need to first define and comprehend the topology:

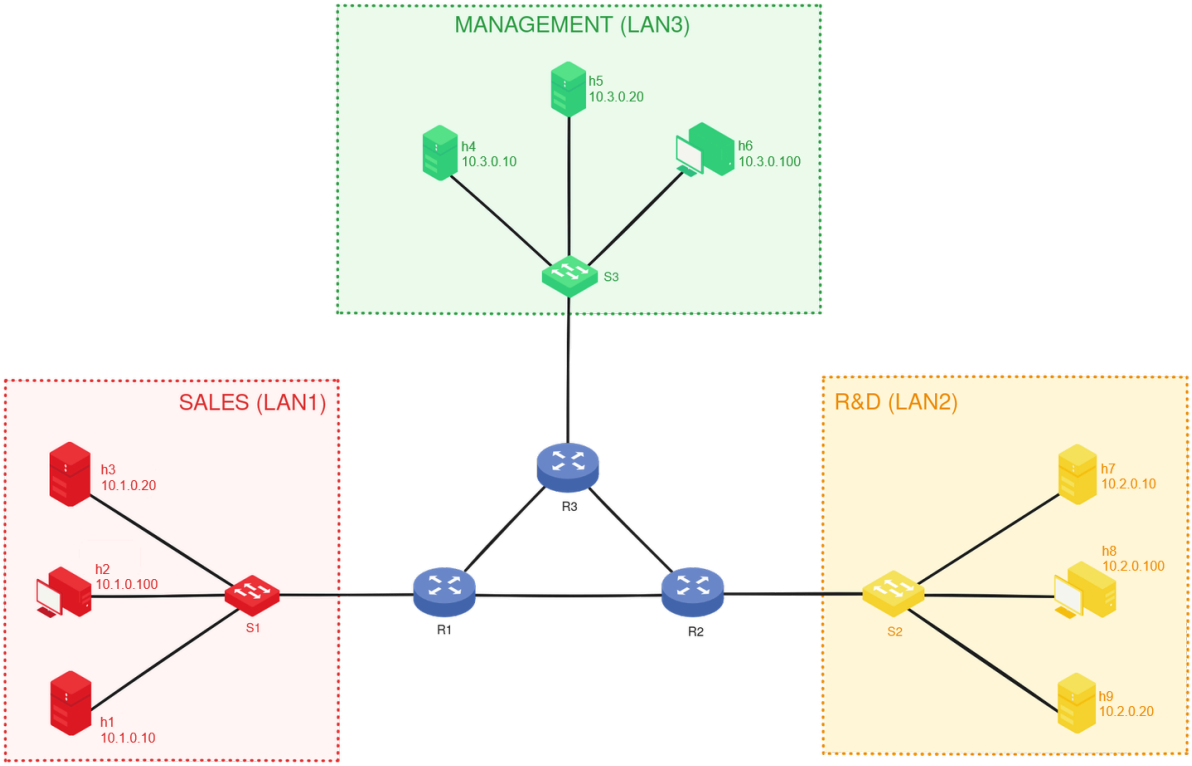

Figure 1: Network topology for TechSecure Inc.

Figure 1, illustrates the topology. In red, representing the **Sales** department we have LAN1 ; in green, representing the **Management** department we see LAN3 ; and in yellow, representing the **Research & Development** department we have LAN2.

In the centre, we have our core network composed of three `BMv2 P4` routers that will handle routing the traffic and applying the firewall rule-set. As for the switches of each `LAN`, we have three `OpenVSwitches`, one for each LAN. `OpenVSwitches` usually need a software controller to switch packets, however, in this project we are configuring them manually through the use of `ovs-ofctl`.

2.2. Network Elements

As previously mentioned in Section 1, to emulate the network we are using `mininet`.

Everything in `mininet` is either a **host** or a **switch**, so to create a router, we need to adapt a switch, adding routing capabilities. Due to `P4`-centric nature of this project, we are using `P4Host` objects as hosts, and `P4Switch` objects as `P4` routers, these are custom implementations of the `Host` and `Switch` classes that provide `P4` integration into the `mininet` topology and can be found in the `p4-utils` repository.

The `OpenVSwitches` are simply `OVSKernelSwitch`-based switches, an internal `mininet` switch implementation that allows the injection of `ovs` rules using either a controller or the `ovs-ofctl` command-line utility.

To allow communication between nodes in the network, `mininet` leverages `MAC` addresses. While we can also define an `IP` address for each node, its use is only for identification. To facilitate the manage-

ment and enhance the intuitive allocation of MAC addresses, we have established a set of guidelines that define the construction of these addresses for any node within the network.

This way, MAC addresses must adhere to the format `00:00:00:{node_type}:{node_id}:{port_id}`, where *node_type* represents the type of the node (1, 2 and 3 for routers, hosts and switches, respectively); *node_id* represents the identifier of the node (3 if the identifier is "h3", for example); and *port_id* represents which port the address corresponds to.

For example, from Figure 1, the host h4 will have the MAC address of `00:00:00:02:04:01`, since it is a host whose identifier is 4 and the port is the number 1.

2.3. Configuring the Topology

The MAC and IP addresses alone are not enough to define a topology, there is the requirement of setting the rules for the firewalls, defining LAN's, defining the nodes and respective links, or even, setting up default values, such as file paths or behavioral models.

Having these values hard-coded on the topology is not only bad practice, as it makes debugging and configuring a lot harder. To amend this issue, we make use of an YAML configuration file. The specification of the configuration file is documented on the README.md file at the project's repository.

The configuration file can be thought of as a text representation of the topology.

3. Executing the Project

Due to the complex file structure, existing configuration files and multiple dependencies, running the work assignment may not be 100% straightforward. The following listing illustrates a set-up guide:

1. Clone the project repository:

```
git clone https://github.com/gweebg/sdn-firewall.git
cd sdn-firewall
```

2. Set-up Python environment and install required packages:

```
python -m venv .env
source .env/bin/activate
pip install -r requirements.txt
```

3. Compile P4 source files into JSON:

```
sh compile-p4.sh # Uses "p4c-bm2-ss"
```

4. Run the topology in mininet:

```
sudo python src/topology --config <config_filepath>
```

After going through each step, without any errors, the project should be running in the terminal with the `mininet CLI` up and running.

4. Routing and Switching Packets

Starting the project, the group was given a base P4 router implementation, so, in terms of building the basic routing logic there was no added effort. All that was needed to do was building the set of rules for each router in order to successfully forward packets through the network.

The base P4 implementation provided an IPv4 and Ethernet parser and deparser, as well as their respective tables and actions for the ingress phase of the pipeline. However, due to the firewall con-

straints defined in Section 2, we also needed a way to handle `TCP` and `UDP` packets, manage sessions and filter traffic based on rules.

In this section we dive into each phase of the `P4` pipeline in detail, explaining and justifying new editions or removals to the original code. Phases not included in this section are not deemed relevant, either because it is not used (checksums) or it was not changed (ingress).

4.1. Parsing and Deparsing

First off, in order to handle the new packets, we need to be able to comprehend them, by adding their states in the parsing phase (Listing 1) and respective deparsing phase:

```
state parse_tcp {
  packet.extract(hdr.ports);  // populate ports header
  packet.extract(hdr.tcp);
  transition accept;
}
```

```
state parse_udp {
  packet.extract(hdr.ports);
  packet.extract(hdr.udp);
  transition accept;
}
```

Listing 1: New states for TCP and UDP packets.

With the two newly added packet types, we need to amend the previous `IPv4` parsing state by adding the new types, as we can in Listing 2. Note that `TYPE_TCP` and `TYPE_UDP` are constant values representing the hexadecimal protocol code used on the `Ethernet` packet.

```
state parse_ipv4 {
    packet.extract(hdr.ipv4);
    transition select(hdr.ipv4.protocol) {
        TYPE_TCP: parse_tcp;
        TYPE_UDP:  parse_udp;
        default: accept;
    }
}
```

Listing 2: Updated IPv4 state with the new packet types.

All that's left to do in these phases (parsing and deparsing) is to add the new entries corresponding to the new packets to the deparser. We can do this by appending new `packet.emit(...)` entries to the deparser file.

4.2. Egress Phase

The egress phase refers to the stage where packet processing happens after the packet has been transmitted through the ingress pipeline, and before it is sent out of the network device. This is the single most important step in the pipeline, since it is where the packet filtering for the firewall is done.

Note that this part of the project could have been implemented on the ingress phase, result-wise nothing would have changed. However, to keep the code clear and organized, we opted for the egress phase since it was not being utilized.

There is one added benefit of doing so, in the case of `P4 Switch` objects being able to process multiple packets in different phases, delegating the filtering process to the egress phase, reduces the probability of heavy workloads on the ingress phase.

To achieve the basis of packet filtering, we have to decide whether to allow or block a packet, we can use a value to represent the decision. In this case, we are using `meta.default_rules_allowed`, a 1 bit size register, whose value represents the decision. If the value is 1, then the packet is allowed, else it is blocked.

Blocking a packet in the context of networks is nothing more than dropping it, on the other hand, allowing a packet is just forwarding it. For the ease of use, we opted to define an action for allowing or blocking a packet, demonstrated in Listing 3:

```
action RulesSuccess(){
    meta.default_rules_allowed = 1;
}
```

```
action RulesBlocked(){
    meta.default_rules_allowed = 0;
}
```

Listing 3: Actions for allowing/blocking packets.

Next, we must construct the table in order to apply the rules defined on the configuration file. The rules are injected automatically by the `mininet Python` script, so at startup every rule will already be on the table.

```
table fwall_rules {
    key = {
        hdr.ipv4.srcAddr : ternary;
        hdr.ipv4.dstAddr : ternary;
        hdr.ipv4.protocol : exact;
        hdr.ports.dstPort : exact;
    }
    actions = {
        RulesSuccess;
        RulesBlocked;
    }
    default_action = RulesBlocked;
}
```

Listing 4: Firewall rules forwarding table.

From Listing 4, we see that for matching the protocol and destination port we use exact match, however for the source and destination addresses we use the ternary algorithm. The use of ternary is necessary since it allows for longest prefix matching and applying more complex rules that can match a range of addresses or protocols.

If no field matches the rules, the default action marks the packet as blocked, being eventually dropped.

This implementation by itself is sufficient, providing simple packet filtering according to pre-defined rules, however how does it behave with returning packets ? For example, say we issue a `TCP` request from `LAN1` to `LAN2`, the firewall at `R1` won't block the packet since it is outbound, but how will it deal with the incoming response ?

4.2.1. State Management & Bloom Filters

The firewall at `R1` will, undoubtedly, block the incoming response. This is a problem, since any packet that "awaits" a response won't receive one. `TCP` requests, due to its connection-oriented nature, is one example, however there are a lot more. We need to make the firewall stateful.

To fix this, we introduced the use of bloom filters and a few more states, besides allow or drop, on this step of the pipeline.

```
register<bit<1>>(BLOOM_FILTER_ENTRIES) bloom_filter;

action drop() { mark_to_drop(standard_metadata); }

action set_needs_fw() { meta.needs_fw = 1; }
```

Listing 5: Firewall rules forwarding table.

In Listing 5, the `bloom_filter` register is initialized to track entries in the bloom filter, where each entry is a single bit. The `drop` action marks packets to be dropped, while the `set_needs_fw` action sets a metadata flag indicating that a packet needs to go through firewall processing.

A bloom filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It can result in false positives (indicating an element is in the set when it is not) but never false negatives (indicating an element is not in the set when it is). To uniquely identify network connections (allowing session management), bloom filters make use of five-tuples, a structure containing the source IP address, destination IP address, source port, destination port and protocol.

```
action setFiveTupleOutbound() {
    meta.tupleToCheck.localAddr = hdr.ipv4.srcAddr;
    meta.tupleToCheck.remoteAddr = hdr.ipv4.dstAddr;
    meta.tupleToCheck.localPort = hdr.ports.srcPort;
    meta.tupleToCheck.remotePort = hdr.ports.dstPort;
    meta.tupleToCheck.protocol = hdr.ipv4.protocol;
}
```

Listing 6: Setting five-tuples for outbound connections

Listing 6 shows an example of setting the five-tuple for an outbound packet, as well as its code structure. As for its counterpart, an inbound packet, we just swap the `srcAddr` with `dstAddr`.

As previously mentioned, to work with bloom filters, we need to be able to write and read from it, determining whether the packet belongs to an existing connection or not. Actions `getRegisterPositions()`, `writeState()` and `readState()` are responsible for managing the bloom filter.

```
action getRegisterPositions(){
    bit<32> bfEntries = (bit<32>)BLOOM_FILTER_ENTRIES;

    hash(meta.register_position_one, HashAlgorithm.crc16,
         (bit<32>)0, meta.tupleToCheck, bfEntrys);

    hash(meta.register_position_two, HashAlgorithm.crc32,
         (bit<32>)0, meta.tupleToCheck, bfEntrys);
}

action writeState(){
    bloom_filter.write(meta.register_position_one, meta.state);
    bloom_filter.write(meta.register_position_two, meta.state);
}

action readState(){
    bloom_filter.read(meta.register_cell_one, meta.register_position_one);
    bloom_filter.read(meta.register_cell_two, meta.register_position_two);
}
```

Listing 7: Firewall rules forwarding table.

The `getRegisterPositions()` action calculates positions within the bloom filter based on the five-tuple of the packet being processed. It uses two different hash functions (`crc16` and `crc32`) to compute these positions, ensuring a more robust distribution. The computed positions are stored in `meta.register_position_one` and `meta.register_position_two`.

The `writeState()` action writes the current state of the connection, stored in `meta.state`, to the bloom filter at the positions calculated previously. This updates the bloom filter to reflect the new state of the connection.

The field `meta.state` can take on specific values that represent different states of a TCP connection. These values are determined based on the flags in the TCP header:

- **0** — Represents a reset state, indicating that the connection has been reset. This state is set when the RST (Reset) flag in the TCP header is valid (`hdr.tcp.rst == 1`).
- **1** — Represents an ongoing connection state. This is the default state set for a connection if neither the RST nor the FIN flags are present in the TCP header. It indicates that the connection is active and ongoing.
- **2** — Represents a finished state, indicating that the connection has been terminated gracefully. This state is set when the FIN (Finish) flag in the TCP header is valid (`hdr.tcp.fin == 1`).

These states are used to track the lifecycle of TCP connections within the bloom filter, allowing the firewall to make informed decisions about whether to allow or block packets based on the current state of the connection.

Finally, the `readState()` action reads the state values from the bloom filter at the calculated positions and stores them in `meta.register_cell_one` and `meta.register_cell_two`. This allows the firewall to determine the current state of the connection by checking the values read from these positions. This sequence ensures that the firewall can accurately track and update the state of network connections using the bloom filter.

4.2.2. General Flow

The bloom filter implementation alone will not automatically manage traffic sessions, there is a lot of logic behind the scenes at the egress phase. To summarize how one handles the states and makes use of the bloom filters, Figure 2 illustrates, as a flow diagram, the steps of the egress phase.

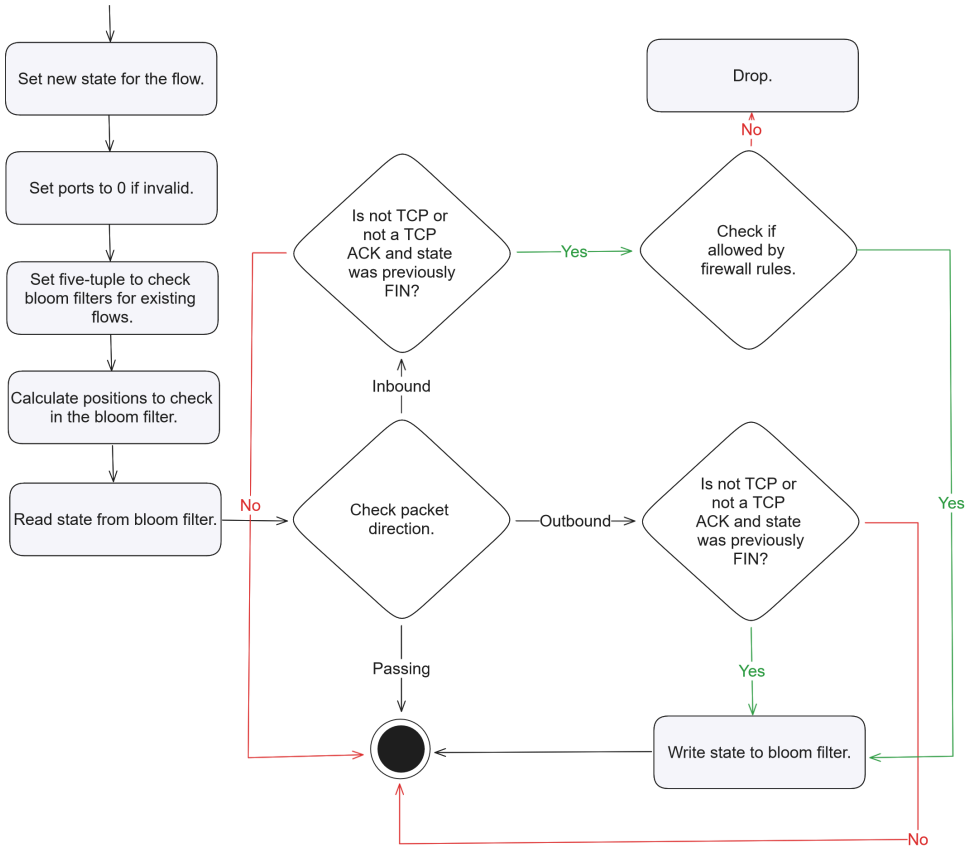

Figure 2: Flow on the egress phase.

We can divide the egress phase into four general steps:

1. The `apply` block in the P4 firewall code executes the main logic for processing packets based on their TCP state and firewall rules. Initially, it sets `meta.state` by checking the TCP flags: if the RST flag is present, the state is set to 0 (indicating a reset); if the FIN flag is present, the state is set to 2 (indicating a finished connection); otherwise, the state is set to 1 (indicating an ongoing connection). If the port headers are invalid, the ports are reset to zero using `setPortsToZero()`.
2. Next, it determines the packet direction using `meta.packetDirection`. If the packet is outbound (`meta.packetDirection == 1`), the five-tuple is set using `setFiveTupleOutbound()`. If the packet is inbound (`meta.packetDirection == 2`), `setFiveTupleInbound()` is called. These actions ensure the correct five-tuple values are used for subsequent operations.
3. The `getRegisterPositions()` action then calculates positions in the bloom filter using hash functions, which are used to read the current state from the bloom filter via the `readState()` action. The state is evaluated to determine if the packet is allowed by checking `isStateAllowed`, which ensures both read positions in the Bloom filter are in the allowed state. Additionally, `isAckAfterFin` checks for specific conditions related to the TCP ACK flag and previously finished connections.
4. For outbound packets, if the packet is not a TCP acknowledgment after a FIN, the current state is written to the bloom filter using `writeState()`. For inbound packets, if the state is not allowed and the packet is not a TCP acknowledgment after a FIN, firewall rules are applied using `fwall_rules.apply()`. If the rules block the packet (`meta.default_rules_allowed == 0`), the packet is dropped. If allowed, or if the state was initially allowed, the state is updated in the Bloom filter. If none of these conditions apply, the packet is processed without further action (`NoAction()`).

This logic ensures that the firewall accurately tracks and enforces connection states and rules, efficiently managing packet flow based on both current connection states and predefined firewall policies.

4.3. Switch Configuration

Moving on from the `P4` aspect of the firewall implementation, we will now briefly discuss how to set up the `OpenVSwitches` without the use of a software controller. Using the `ovs-ofctl` binary, we can manually define or remove rules for a switch.

However, we don't even need to manually define each rule for each address, since there is available the action "`normal`" that handles packets using its built-in, traditional `MAC`-learning switch logic, similar to how conventional `Ethernet` switches operate.

When this action is applied, the switch maintains a `MAC` address table, learns the source `MAC` addresses of incoming packets, and updates its table accordingly. For packet forwarding, it looks up the destination `MAC` address in the table to determine the appropriate output port. If the destination `MAC` address is unknown, the switch floods the packet to all ports except the one it arrived at.

This action ensures compatibility with traditional network switches and simplifies configuration by automating `MAC` learning and packet forwarding, though it may be less efficient and offer less granular control compared to specific flow entries.

To apply the "`normal`" to a switch, we used the following command: `ovs-ofctl add-flow {switch_name} action=normal`, on each switch. The action is applied automatically at start up by the topology `Python` script.

5. Experimental Analysis

In this section, we will prove the correct functioning of the developed project using the topology described in Section 2 and the firewall rules described in Section 2, as well.

As previously stated, the rules are defined on a `YAML` configuration file, so their format will not be the common format found in the usual `commands.txt` file, usually directly injected onto the routers after on start up. Nevertheless, we will provide both formats, for easier comprehension.

```yaml
# Router 1
rules:
  - srcIp:
      network: 3
      host: 0
      mask: 24
    dstIp:
      network: 1
      host: 10
      mask: 32
    protocol: "0x06"
    port: 443
  - srcIp:
      network: 2
      host: 0
      mask: 24
    dstIp:
      network: 1
      host: 20
      mask: 32
    protocol: "0x06"
    port: 25
```

```yaml
# Router 2
rules:
  - srcIp:
      network: 1
      host: 0
      mask: 24
    dstIp:
      network: 2
      host: 10
      mask: 32
    protocol: "0x06"
    port: 80
  - srcIp:
      network: 3
      host: 0
      mask: 24
    dstIp:
      network: 2
      host: 20
      mask: 32
    protocol: "0x06"
    port: 22
```

```yaml
# Router 3
rules:
  - srcIp:
      network: 1
      host: 0
      mask: 24
    dstIp:
      network: 3
      host: 10
      mask: 32
    protocol: "0x06"
    port: 8080
  - srcIp:
      network: 2
      host: 0
      mask: 24
    dstIp:
      network: 3
      host: 20
      mask: 32
    protocol: "0x06"
    port: 443
```

Listing 8: Firewall rules for Section 2's scenario.

Listing 8 illustrates the rules for routers 1, 2 and 3, respectively. We can see each rule being defined as a source address, a destination address, the protocol to allow and the respective port ports.

Each address is composed of a *network*, *host* and *mask* fields as an address, in this project, follows the structure of `10.{network}.0.{host}/{mask}`, for example the destination address of the first rule on router 1 would be represented as `10.1.0.10/32`. The *mask* field defines either if we want an exact or "broad" match of the address, for example, allowing all the traffic from a `LAN` versus allowing only TCP traffic on port 443 on a `LAN`.

To convert this format into a format understandable by `P4` we just need to follow the template:

```
table_add fwall_rules RulesSuccess {srcIp} {dstIp} {protocol} {port} 1 1
```

Keeping in mind that both *srcIp* and *dstIp* are converted to a string representing their complete ternary format.

5.1. Test Suite

As for the test suite used to test the project, each firewall rule is tested with *netcat*. The test is simple and straighforward, for example, on the first rule of router 1 (Listing 8), we want to test if we can connect any host (the mask defines if we're testing one or more hosts) from `10.3.0.0/24` to the host `10.1.0.10` via TCP (`0x06`) on port 443.

The Management LAN, represented by the address `10.3.0.0/24`, is composed by hosts 4, 5 and 6. The Sales LAN, represented by `10.1.0.0/24` contains our host test subject `10.1.0.10`, identified by the number 1.

So, for each one of the Management LAN hosts, we try to reach the host 1 from the Sales LAN via TCP on the accepted port (in this case 443), as well as on a blocked port. This way, we guarantee that we are both correctly allowing and blocking traffic.

```
==================Testing rule from: 10.3.0.0/24 to: 10.1.0.10/32 with proto: 0x06 on port: 443====================
h1 running nc server: nc -nl 443  -v

Testing h4 <-> h1:
    h4 running wrong command: nc 10.1.0.10 444  -w 2 -v -> connection timed out correctly
    h4 running correct command: nc -z 10.1.0.10 443  -w 2 -v

Testing h5 <-> h1:
    h5 running wrong command: nc 10.1.0.10 444  -w 2 -v -> connection timed out correctly
    h5 running correct command: nc -z 10.1.0.10 443  -w 2 -v

Testing h6 <-> h1:
    h6 running wrong command: nc 10.1.0.10 444  -w 2 -v -> connection timed out correctly
    h6 running correct command: nc -z 10.1.0.10 443  -w 2 -v

h1 command result:
   Listening on 0.0.0.0 443
   Connection received on 10.3.0.10 34886
   Connection received on 10.3.0.20 46480
   Connection received on 10.3.0.100 57124
====================h1 nc server stopped====================
```

Figure 3: Results of the test for the first rule of R1.

Figure 3 illustrates the result of the test for the first rule of R1. The testing is automatic, meaning that each and every rule is tested at start up. If any rule is showing invalid results, the execution is aborted.

This process repeats for each rule defined in the configuration file, ensuring that the firewall is executing successfully.

6. Final Remarks

Closing this report, the group agrees that each and every challenge proposed for this work assignment has been correctly met. Building the basic firewall system logic, was easier than expected, however, the stateful part threw us a curveball, and it was definitely the hardest to both research and develop.

In future work, we expect to be adding support for `ICMP` packets, network address translation (`NAT`), and a generic load balancing mechanism to improve performance and scalability of the network.