4156 Final Project Report

Programmable P4 Switch Firewall for IOT System

December 22, 2022

Abstract

In this project, we built a firewall on the switch level by P4 language for the IOT system. A demo is constructed based on 3 host machines and 3 switches in the mininet. The background of general firewall will be presented firstly, followed with P4 Firewall and its implementation; The demo part will be shown afterward, the block mechanism also will be illustrated. The last section is the future work and limitations to be improved.

1 Background

Cybersecurity is one of the major concerns around the computer science area especially recent years. It is reported that the expected cost will be surged from $8.44 trillion in 2022 to $23.84 trillion in 2027 (Statista, 2022). The huge cost makes the security expert aware of the effort to block the malicious address requesting like building the network firewall. A few types of firewall are listed here to provide the background of the firewall.

- Layer 3 IP-Layer Filter, this is the most typical firewall at the network layer, it technically works as monitoring and filtering the harmful traffic by the same protocols defined at the routers.
- Session Layer Firewall, this layer will use the circuit-level gateway to verify the established TCP and keep track of the sessions, the session would be disconnected when the sniffer detect the malicious traffic.
- Application Layer Firewall, it is known as the proxy firewall, the packet request instead accessing the internal service directly, it will be sent through the proxy firewall firstly, the malicious request will be blocked at the proxy firewall.

2 P4 Firewall

The P4 firewall at the switch level is working as packet filtering, once the table rule is defined at the ingress stage, the packet will be blocked based on the source and destination IP address. Since P4 firewall at the switch level is also the hardware level, the packet will also combine the ethernet address and switch actions. There are several benefits of P4 firewall:

- Scalability: the P4 provides the control plane to program the table routing rule, therefore, we only need to define once the control plane, it should work for all the devices at the data plane.
- Cost Saving: Compared with proxy firewall, P4 does not require adding another application like proxy server by VPN/VPS at the application layer.
- Reliability: since switch is a centralized and necessary part of the network hardware, the firewall will be located at Data Link layer, the low level firewall is more reliable than a higher level.
- Programmability: the traditional switch does not provide the firewall functionality, P4 makes it possible to build up a firewall.

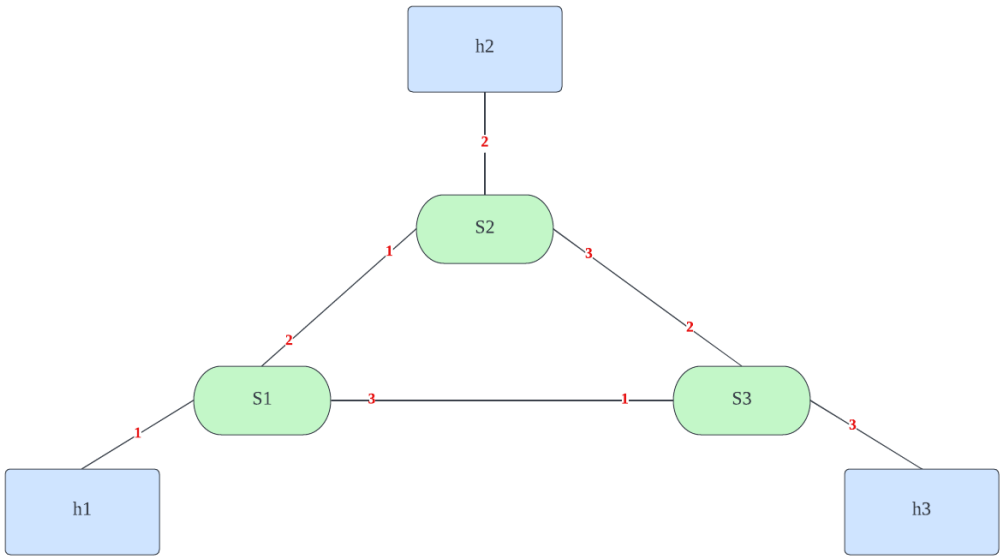

Figure 1: A 3 by 3 switches and host machines graph

3 Demo Pipeline

3.1 Topology Graph

In this topology graph, we have three hosts h1 (10.0.1.1), h2 (10.0.2.2), h3 (10.0.3.3) and three switches s1, s2, s3. Assuming we are building a firewall at S1 to protect h1, and h2 is a malicious site. The workflow is basically to block h2 and check:

1. Whether h2 can receive messages from h1.
2. The firewall table rules of switch s1 and s2.

3.2 Demo Files

pod-topo	update readme	3 weeks ago
utils	reconstructed topo and fixed connection issue	2 weeks ago
.gitignore	update readme	3 weeks ago
LICENSE	Initial commit	last month
Makefile	init basic tunnel	last month
README.md	Update README.md	2 weeks ago
basic.p4	reconstructed topo and fixed connection issue	2 weeks ago
block.py	finish all experiments	2 weeks ago
controller.py	finish all experiments	2 weeks ago
myTunnel_header.py	reconstructed topo and fixed connection issue	2 weeks ago
receive.py	reconstructed topo and fixed connection issue	2 weeks ago
send.py	finish all experiments	2 weeks ago

Figure 2: Files under the project folder

There are several essential files we used in the project from 2.

- controller.py: build p4Runtime table rules by program, and print switch s1 connection every 2 seconds.
- send.py: used by h1, to send packet to h2 and h3 at xterm terminals (bind with h1,h2,h3 ip address).
- receive.py: used by h2,h3 to check the whether packet sent from h1 can be received.
- block.py: send a signal to P4Runtime to block h2/h3 at s1.

3.3 P4Runtime Example

We constructed the topology in P4Runtime instead of hardcode in the json file. An example piece of code is shown as following:

```
# setup the connection for s1
writeTableRules(p4info_helper, ingress_sw=s1, dst_eth_addr="08:00:00:00:01:11", dst_ip_addr="10.0.1.1", port=1, dst_id=100)
writeTableRules(p4info_helper, ingress_sw=s1, dst_eth_addr="08:00:00:00:02:22", dst_ip_addr="10.0.2.2", port=2, dst_id=200)
writeTableRules(p4info_helper, ingress_sw=s1, dst_eth_addr="08:00:00:00:03:33", dst_ip_addr="10.0.3.3", port=3, dst_id=300)

# setup the connection for s2
writeTableRules(p4info_helper, ingress_sw=s2, dst_eth_addr="08:00:00:00:01:11", dst_ip_addr="10.0.1.1", port=1, dst_id=400)
writeTableRules(p4info_helper, ingress_sw=s2, dst_eth_addr="08:00:00:00:02:22", dst_ip_addr="10.0.2.2", port=2, dst_id=500)
writeTableRules(p4info_helper, ingress_sw=s2, dst_eth_addr="08:00:00:00:03:33", dst_ip_addr="10.0.3.3", port=3, dst_id=600)

# setup the connection for s3
writeTableRules(p4info_helper, ingress_sw=s3, dst_eth_addr="08:00:00:00:01:11", dst_ip_addr="10.0.1.1", port=1, dst_id=700)
writeTableRules(p4info_helper, ingress_sw=s3, dst_eth_addr="08:00:00:00:02:22", dst_ip_addr="10.0.2.2", port=2, dst_id=800)
writeTableRules(p4info_helper, ingress_sw=s3, dst_eth_addr="08:00:00:00:03:33", dst_ip_addr="10.0.3.3", port=3, dst_id=900)
```

Figure 3: Topology in P4Runtime

For instance, the second line of code in 3 defines the connection between s1 (switch 1) and h1 (10.0.1.1).

4 Block Mechanism

Figure 4: Block TableEntry Workflow

The block runs as firstly executing block.py to send h2 as signal to P4Runtime.py execution binary by UDP socket, then P4Runtime.py receives the signal on the port and call the function BlockTableEntry and do the block work.

As it shown in 5, the block mechanism is a two step blocking, the first step is to construct the tableRule matches in the topology and then call the runtime library to delete this entry. After that, a new insertion of the table will be called to write the table rule with the action_name defined as drop.

```python
def blockTableEntry(p4info_helper, ingress_sw, dst_eth_addr, dst_ip_addr, port, dst_id):
    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.ipv4_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
        },
        action_name="MyIngress.ipv4_forward",
        action_params={
            "dstAddr": dst_eth_addr,
            "port": port,
            "dst_id": dst_id
        }
    )
    ingress_sw.DeleteTableEntry(table_entry, False)

    # Change to drop func
    table_entry = p4info_helper.buildTableEntry(
        table_name="MyIngress.ipv4_lpm",
        match_fields={
            "hdr.ipv4.dstAddr": (dst_ip_addr, 32)
        },
        action_name="MyIngress.drop"
    )
    ingress_sw.WriteTableEntry(table_entry)
```

Figure 5: 2-Step Block Table Entry

5 Demo Video

Demo Video can be found at https://youtu.be/Ub2DVaiTNm0.

6 Future Work & Limitations

Several improvements can be worked through by:

1. A real-time monitor at s1 is ideal, when accessing s2 the block.py is triggered rather than manually.
2. The implementation of blocking is 2-steps (delete + add) rather than updating the table directly by runtime. We tried to implement the UpdateEntryRule but the "repetition for the same destination ip address" is raised, need more research work on the low level bmv2 model of P4Runtime.
3. The topology is manipulated by mininet, a real network with servers&switch would be great.
4. A subnet block does not support in the project, when trying to block 10.0.2.2/16, the error AssertionError is raised from the library convert.py.