Corrupting P4 programs by manipulating packet data

ALENA SHCHEGLOVA
Supervisor(s): FERNANDO KUIPERS, CHENXING JI
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Paper Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Data planes are responsible for forwarding packets in a network. The P4 language is used for programming programmable data planes. Such data planes give more flexibility to programmers by allowing them to define how the packets should be processed. However, these data planes might also be more vulnerable to malicious attacks than traditional (non-programmable) data planes. That is because software is usually more prone to errors as compared to the hardware. Different research has already analyzed various aspects of the security of the P4 language. However, the security vulnerabilities of P4 programs have not been researched in depth. The main contribution of this paper is providing examples of attacks on P4 programs by using manipulated packet data. In this research, it was attempted to corrupt three P4 programs by manipulating packet data. Two of the three attempts were successful. The paper concludes that some P4 programs can be corrupted by malicious packets.

1 Introduction

In a network, devices like routers and switches usually consist of a control plane and a data plane. The control plane determines which path to use to send a packet. The data plane is responsible for forwarding packets. Traditionally, the functionality of data planes is predetermined by the manufacturer. So, once the network device is finished there is no easy and cheap way to change its functionality. However, programmable data planes have changed that. Using languages like P4 [1], programmers are now able to define how the packets are going to be processed, without being restricted to the functionality implemented by the manufacturer. However, the flexibility of the programmable data planes may also cause them to be more susceptible to malicious attacks. Since P4 is a relatively new language, there has not yet been extensive research into its security.

Previous research provides some insight into the security vulnerabilities of the P4 language as well as the programmable data planes. In [2], a STRIDE analysis of a P4 platform was presented. STRIDE is a model for categorizing IT-security threats. The researchers described potential attacks, and vulnerabilities related to the P4 language, compiler, the controller, the P4 Runtime, as well as the switches. In [3], the researchers automated the attack discovery of a certain class of attacks on data planes. They provided an example of such an attack and its discovery. In [4], assertion-based verification was used to check the general security and correctness properties of several P4 programs. The analysis was based on translating a P4 program into a C program. The C model was verified by a symbolic engine. The researchers analyzed four P4 programs and managed to find bugs in each of them. In [5], a few buggy P4 programs were produced. The researchers analyzed how running these programs on a switch affected the work of the switch.

This research attempts to answer the question: "Can the attacker manipulate the packet data to corrupt certain P4 programs?". To answer this question, first, three P4 programs were analyzed and the vulnerable fields of these programs were discovered. After that, the fields in the packets, that could corrupt these P4 programs, were identified. We used previously discovered vulnerabilities along with manipulated packets to demonstrate attacks on the chosen P4 programs.

The main contribution of this paper is providing examples of attacks on P4 programs by using packets with manipulated data. These attacks aimed at corrupting the behavior of the P4 programs.

The rest of the paper will have the following structure. Section 2 describes the methodology used for this research. It is followed by Section 3 with the description of the experimental setup and results. Section 4 contains a reflection on an ethical aspect of the research. Finally, the conclusion is presented in Section 5.

2 Methodology

The first step in the research was a literature study. During the literature study examples of vulnerabilities in buggy P4 programs were found, as well as examples of attacks on programmable data planes. Afterwards, three P4 programs were chosen to be analyzed:

- load_balance.p4 [6]
- mri.p4 [7]
- firewall.p4 [8]

These programs were chosen with reproducibility in mind. As they are part of the official P4 tutorial [9] and have open source code, everyone can get access to them and easily work with them.

Load_balance.p4 is an implementation of load balancing. The load_balance.p4 uses a hash function to determine which one of the two hosts the packet will be forwarded to.

Mri.p4 allows users to track the path and the length of queues that every packet travels through. It appends an ID and queue length to the header stack of every packet.

Firewall.p4 implements a simple stateful firewall. It allows hosts in the same internal network to communicate with each other and with the hosts from the external networks. However, it prevents hosts from an external network to initiate communication with hosts from the internal network.

To analyze the programs, we went over each program, understood its control flow, what it was doing, and detected potential vulnerabilities. After analyzing and identifying the vulnerable fields of the programs, we tested if these vulnerabilities could, indeed, be exploited by a malicious data packet. For packet manipulation, Scapy [10] was used. Scapy is a python library that allows to create/send/sniff data packets. To run the P4 programs a bmv2 switch [11] was used. In order to be able to send packets between the switches, we used Mininet [12]. Mininet is a network emulator that allows to create typologies of hosts/switches and send packets between them.

3 Experimental Setup and Results

To set up the experiments, it is necessary to follow the steps described in the P4 tutorial [9]. The tutorial explains how to

set up necessary software, install dependencies and run the P4 programs. We used load_balance.p4 [6], mri.p4 [7] and firewall.p4 [8] programs from the tutorial. All P4 programs are run on bmv2 switch [11]. For each P4 program, the tutorial provides a Mininent network as well as the python scripts that allow to create/manipulate/sniff packets and send them between the hosts in the Mininet network.

The following experiments used only the tools provided by the tutorial and only some changes to the python scripts were made. The changes will be described in each experiment.

load_balance.p4

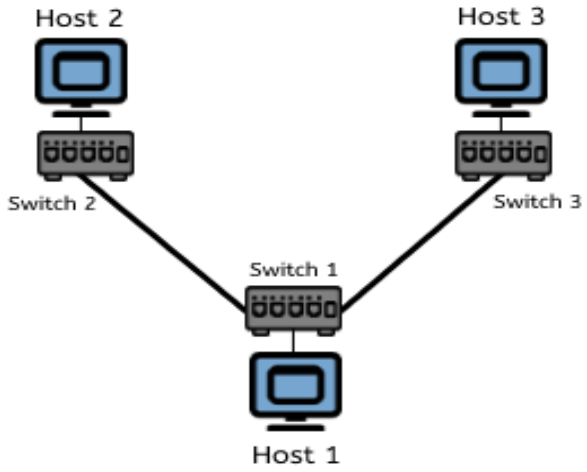

Figure 1: Load balance topology

The load_balance.p4 uses a hash function to forward the packets to one of the two hosts. The purpose of a load balancer is to distribute traffic in order to avoid overloading one of the two hosts. For this experiment, a topology from Fig. 1 was used. In the topology, there are three hosts, each connected to its own switch. When host 1 sends packets, switch 1 forwards them either to host 2 or host 3.

We discovered through analysis that the simple hash function used in the program is a vulnerability that can be exploited by an attacker. In order to select one of the two hosts, the hash function is applied to a 5-tuple consisting of the source and destination IP addresses, IP protocol, and source and destination TCP ports. This means that two packets with identical source and destination IP addresses, IP protocol, and source and destination TCP ports will always have the same hash value and will go to the same host (either host 2 or host 3).

First, we ran this P4 program with the given send.py file. This python script generates packets (using Scapy [10]) with a randomized TCP source port. We only performed a slight change to the given script, by moving the code that sends a packet into a loop with the range from 0 to 50. This allowed us to send 50 packets at once. When we sent 50 packets from host 1, the packets got evenly distributed between the two destination hosts (Fig. 2). We repeated that several times and the packets were always evenly (almost) distributed between the two hosts. Thus, we observed the intended behavior of load_balance.p4.

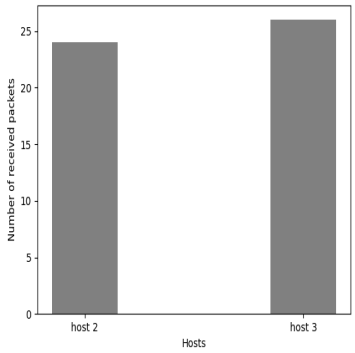

Figure 2: Intended behavior of load_balance.p4

To make use of the hash function vulnerability, it is necessary to change one of the values of a 5-tuple and see how it affects where the packet is being sent. Since the initial python script randomly chooses the TCP source port, we decided to focus on this field of the packet. So, we again sent the traffic of 50 packets and recorded which ones got sent to host 2. We took the TCP source port values of the 20 packets that got received by host 2 and put those values in an array. Then we set a loop and sent the 20 packets (Fig. 3) with the TCP values we had previously put into an array. Now, as expected, the load balancer did not divide the traffic but instead sent everything to h2 leaving h3 idle (Fig. 4). We repeated that several times with various TCP source ports and various amounts of packets - the result was always the same.

h1 -> h2 IpSrc, IpDst, TcpSport, TcpDport	
10.0.1.1, 10.0.0.1, 52220, 1234	10.0.1.1, 10.0.0.1, 49733, 1234
10.0.1.1, 10.0.0.1, 61807, 1234	10.0.1.1, 10.0.0.1, 53853, 1234
10.0.1.1, 10.0.0.1, 60297, 1234	10.0.1.1, 10.0.0.1, 51917, 1234
10.0.1.1, 10.0.0.1, 58318, 1234	10.0.1.1, 10.0.0.1, 49788, 1234
10.0.1.1, 10.0.0.1, 49968, 1234	10.0.1.1, 10.0.0.1, 57047, 1234
10.0.1.1, 10.0.0.1, 50252, 1234	10.0.1.1, 10.0.0.1, 64762, 1234
10.0.1.1, 10.0.0.1, 49556, 1234	10.0.1.1, 10.0.0.1, 50742, 1234
10.0.1.1, 10.0.0.1, 64872, 1234	10.0.1.1, 10.0.0.1, 58200, 1234
10.0.1.1, 10.0.0.1, 49711, 1234	10.0.1.1, 10.0.0.1, 60024, 1234
10.0.1.1, 10.0.0.1, 51593, 1234	10.0.1.1, 10.0.0.1, 52238, 1234

Figure 3: Packets that got sent to h2

Hence, we managed to disrupt the work of load_balance.p4 and prevent it from distributing the traffic between the two hosts. The danger of such an easily manipulated hash function is that an attacker can overload one of the hosts, which can lead to a DoS attack of that host.

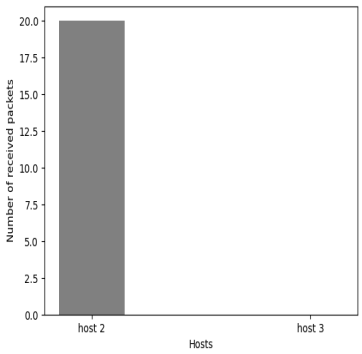

Figure 4: Corrupted behavior of load_balance.p4.

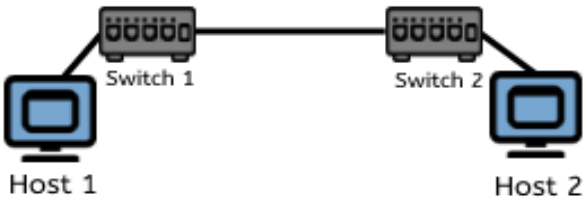

Figure 5: Mri topology

mri.p4

Mri.p4 allows users to track the path and the length of queues that every packet travels through. It does that by appending an ID and queue length to the header stack of every packet. So, since the mri.p4 adds a header to the packet it processes, this gave an idea to recreate a buffer overflow attack, described in [13]. In that paper, however, the data plane was programmed in C rather than in P4 language.

For this experiment, a topology from Fig. 5 was used. The idea of the attack was to send a packet of a large size to the switch. The switch would add a header to the packet. This would make the packet larger than the allowed size, which would lead to an overflow and the crash of the system.

We used the provided send.py script to attempt the attack. The only change was made to the load of the packet. To increase the size of the packet, we increased the load of the packet (Fig.6). By trial and error, we figured out that the load of 'X'*1468 was the largest that the packet could have. Anything larger would make the packet exceed the allowed maximum size and hence would prevent it from being sent.

```
pkt = Ether(src=get_if_hwaddr(iface), dst="ff:ff:ff:ff:ff:ff") / IP(
    dst=addr, options = IPOption_MRI(count=0,
        swtraces=[])) / UDP(
        dport=4321, sport=1234) / ('X'*1468)
```

Figure 6: Code change

Once the send.py script was changed, we tried to send the packet from h1 to h2. We expected switch 1 to crash after it adds a header to the packet. Unfortunately, this did not happen and the packet was just dropped by the switch.

After exploring the cause of the outcome, we found out that such an attack is not possible. It is because, as described in [5], the P4 program cannot directly control the next instruction to be executed, since the control flow is immutable. As the control flow is immutable, the buffer overflow attack is not feasible.

firewall.p4

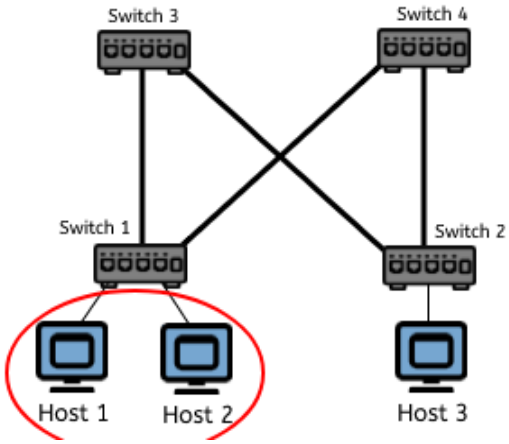

Figure 7: Firewall topology

For this experiment, a topology from Fig.7 was used. Switch 1 is configured with a P4 program that implements a simple stateful firewall (firewall.p4). The rest of the switches implement basic L2-forwarding, which simply forwards all the packets to their destination.

The firewall on switch 1 has the following functionality:

- Hosts h1 and h2 are on the internal network and can always connect to one another. They can also connect to the switch on the external network - h3.
- Host h3 can only reply to connections once they have been established from either h1 or h2, but cannot initiate new connections to hosts on the internal network.

The bloom filter with two hash functions is used to check if a packet coming into the internal network is a part of an already established TCP connection. Two different register arrays are used for the bloom filter; each is updated by a hash function. The hash functions use hash algorithms crc16 and crc32. The hash functions for the packets that come from the internal network are computed on the packet 5-tuple:

(IP.src, IP.dst, TCP.src, TCP.dst, IPv4 protocol type)

And the hash functions for the packets that come from an external network are computed on the packet 5-tuple:

(IP.dst, IP.src, TCP.dst, TCP.src, IPv4 protocol type)

The computed hash values are the bit positions in the two register arrays of the bloom filter (regOne and regTwo).

When h1 (or h2) sends a packet, the firewall computes the values of the two hash functions and uses the results as indices (regOne and regTwo) for the two register arrays. Then it assigns 1 to the regOne element in the first array and 1 to the regTwo element in the second array. When h3 sends a packet to h1 (or h2), the hash functions are computed again, providing indices regOne and regTwo. If the first array does not contain 1 at position regOne or the second array does not contain 1 at regTwo - the packet gets dropped. Otherwise, the packet is considered a reply to the connection of h1 (or h2) and gets forwarded to the host.

However, both hash functions can be manipulated. For that, it is necessary to find at least two packets whose IP and TCP values will result in a collision for both hash functions. When manipulation is successful, h3 can initiate communication with the hosts from the internal network.

We discovered the collisions by recording the register array values for the packets that were sent from h1 to h3 as well as the packets that were sent from h3 to h1. Examples of packets that cause collisions can be found in Fig.8.

For example, h1 can send the following packet to h3 :

IP.src = 10.0.1.1, IP.dst = 10.0.3.3, TCP.sport = 50218 , TCP.dport = 1234.

Then, as intended by the firewall.p4, h3 can reply with the following packet:

IP.src = 10.0.3.3, IP.dst = 10.0.1.1, TCP.sport = 1234 , TCP.dport = 50218.

However, there is another packet that h3 can successfully send to h1:

IP.src = 10.0.3.3, IP.dst = 10.0.1.1, TCP.sport = 428 , TCP.dport = 1234.

That is because both hash functions give the same values for this packet and the first packet (sent from h1 to h3), namely regOne = 2660, regTwo = 3522. Hence, this results in h3 establishing a new TCP connection with h1, which disrupts the intended behavior of the firewall.p4.

h1 -> h3 IpSrc, IpDst, TcpSport, TcpDport	**h3 -> h1** IpSrc, IpDst, TcpSport, TcpDport	**regOne**	**regTwo**
10.0.1.1, 10.0.3.3, 50218 , 1234	10.0.3.3, 10.0.1.1, 428 , 1234	2660	3522
10.0.1.1, 10.0.3.3, 49794 , 1234	10.0.3.3, 10.0.1.1, 624 , 1234	2764	2440
10.0.1.1, 10.0.3.3, 634 , 1234	10.0.3.3, 10.0.1.1, 1806 , 1234	3069	3867
10.0.1.1, 10.0.3.3, 65197 , 1234	10.0.3.3, 10.0.1.1, 63696 , 1234	2964	871
10.0.1.1, 10.0.3.3, 63763 , 1234	10.0.3.3, 10.0.1.1, 33811 , 1234	773	1758
10.0.1.1, 10.0.3.3, 63493 , 1234	10.0.3.3, 10.0.1.1, 64268 , 1234	2876	1837

Figure 8: Packets that cause collisions

4 Responsible Research

The research is based on open sources. All the tools and methods that were used in the research are described in Section 2. The setup of the experiments is presented in detail in Section 3. Since the analyzed programs have open source code, everyone has access to them. Hence, research is reproducible and anyone can repeat the same steps and get the same results. All the papers and resources are mentioned in the relevant parts of the text and are provided in the reference section.

The P4 programs that were analyzed in this research are part of a tutorial and are not used in real life. Thus, there is no risk of making use of the vulnerabilities discovered in these programs.

5 Conclusions

In the network devices, data planes are responsible for forwarding packets. Traditional data planes only allow for predefined protocols, however, that has changed since the emergence of programmable data planes. Programmable data planes allow programmers to define their own protocols and how packets are going to be handled. To program data planes, the P4 language [1] is used. It is a relatively new language and even though there has been some research done concerning the security of P4 [2-5], it has not been studied enough yet.

The question that this paper aimed to answer is “Can an attacker corrupt certain P4 programs by manipulating packet data?”. In order to answer this question, three P4 programs were chosen for analysis: load_balance.p4 [6], mri.p4 [7], firewall.p4 [8]. The analysis helped to identify the vulnerabilities of these programs. After that, the packets were created in such a way that they could make use of the vulnerabilities and corrupt the P4 programs. Hash functions were the vulnerabilities of load_balance.p4 and firewall.p4 programs. To corrupt such programs, malicious packets were created with manipulated TCP headers. Both of the attacks succeeded. For mri.p4 a buffer overflow attack was attempted. To do that, the size of the packet was increased by increasing the size of the load of the packet. However, the attack was unsuccessful and it was not possible to corrupt the P4 program.

Although one of the three attacks was not successful, the paper proves that it is possible to corrupt certain P4 programs by manipulating packet data. With regard to future work, more P4 programs can be analyzed.

References

[1] P4 language. (2022). *P4 Open Source Programming Language* [Online]. Available: https://p4.org/.

[2] A. Agape, M. C. Danceanu, R. Hansen, S. Schmid, ”Charting the Security Landscape of Programmable Dataplanes”, [Online], Jun 30 2018. Available: https://arxiv.org/abs/1807.00128.

[3] Q. Kang, J. Xing, A. Chen. ”Automated attack discovery in data plane systems”, in *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET19)*, Santa Clara, CA, 2019. Available: https://www.usenix.org/biblio-3061.

[4] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho. ”Uncovering Bugs in P4 Programs with Assertion-based Verification”, in *Proceedings*

of the Symposium on SDN Research, LOS ANGELES, CA, USA, 2018, pp. 1-7. Available: https://dl.acm.org/doi/10.1145/3185467.3185499.

[5] M. V. Dumitru, D. Dumitrescu, C. Raiciu. "Can we exploit buggy P4 programs?", in *Proceedings of the Symposium on SDN Research*, San Jose, CA, USA, 2020, pp. 62-68. Available: https://doi.org/10.1145/3373360.3380836.

[6] Load_balance.p4. (2018). *Load balancing* [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises/load_balance.

[7] Mri.p4. (2018). *Multi-Hop Route Inspection* [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises/mri.

[8] Firewall.p4. (2018). *Firewall* [Online]. Available: https://github.com/p4lang/tutorials/tree/master/exercises/firewall.

[9] P4 tutorial (2018). *P4 Tutorial* [Online]. Available: https://github.com/p4lang/tutorials.

[10] Scapy's documentation (2022). *Scapy's documentation* [Online]. Available: https://scapy.readthedocs.io/en/latest/.

[11] Bmv2. *Behavioral model (bmv2)* [Online]. Available: https://github.com/p4lang/behavioral-model.

[12] Mininet (2022). *Mininet* [Online]. Available: http://mininet.org/.

[13] D. Chasaki, T. Wolf. "Attacks and Defenses in the Data Plane of Networks", in *IEEE Transactions on Dependable and Secure Computing*, 2012, vol. 9, no. 6, pp. 798-810. Available: https://ieeexplore.ieee.org/document/6231636.