Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Hausarbeit

Milena Hippler

Finding Vulnerabilities of a SDN Controller by Impersonating a Switch

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Milena Hippler

Finding Vulnerabilities of a SDN Controller by Impersonating a Switch

Hausarbeit eingereicht im Rahmen des Master Hauptprojekts

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Korf

Eingereicht am: 02. Mai 2023

Contents

List of Figures v

List of Abbreviations vi

1 Introduction 1
1.1 Structure . . . 1

2 SDN Networking 3
2.1 SDN Controller . . . 4
2.2 SDN Switch . . . 5
2.3 OpenFlow Communication . . . 5

3 Possibilities of Impersonating a Switch 10
3.1 Existing Frameworks - DELTA . . . 11
3.2 Self-Written Program . . . 13
3.3 Hybrid Solution . . . 14

4 Test Setup 16
4.1 Test Components . . . 16
4.1.1 ONOS Controller . . . 16
4.1.2 Self-Written Program . . . 17
4.1.3 DELTA . . . 21
4.2 Test Environment . . . 22

5 Test Cases 25
5.1 DELTA Attacks . . . 25
2.1.010 – Malformed Version Number . . . 25
2.1.020 – Corrupted Control Message (Corrupted Content) . . . 26
2.1.030 – Handshake without Hello . . . 27
2.1.040 – Control Message before Hello . . . 28
2.1.050 – Multiple Main Connection Request from Same Switch . . . 28
5.2 Self-Written Program . . . 29
#0 - Initial Test . . . 30
#2.1.020 – Corrupted Control Message (Corrupted Content) . . . 30

#3.1.010 – Packet-in Flooding 31

6 Test Results **34**
6.1 Result by Test Case 35
2.1.010 – Malformed Version Number 35
2.1.020 – Corrupted Control Message (Corrupted Content) 37
2.1.030 – Handshake without Hello 39
2.1.040 – Control Message before Hello 39
2.1.050 – Multiple Main Connection Request from Same Switch . . . 39
#0 - Initial Test 41
#2.1.020 – Corrupted Control Message (Corrupted Content) 42
#3.1.010 – Packet-in Flooding 43
6.2 Result Overview 44

7 Conclusion **46**
7.1 Perspective 47

List of Figures

2.1 SDN Architecture [18] 4

3.1 DELTA Architecture [3] 11

4.1 Example Imports 17
4.2 Main with set connection 18
4.3 Start communication 18
4.4 Send message 19
4.5 Thread-based approach 19
4.6 Reply thread 20
4.7 Handle request 21
4.8 DELTA test environment 22
4.9 Test setup structure 22
4.10 Table structure topology 24

5.1 OpenFlow header [7] 26
5.2 Complete Hello handshake [28], [29], [10] 27
5.3 Packet_In 33

6.1 Test cases 34
6.2 Attack started via DELTA 35
6.3 Channel Agent log 36
6.4 Agent Manager log 36
6.5 Test performed on the table setup 37
6.6 Test with corrupted message control 38
6.7 Agent Manager log 38
6.8 Agent Manager log 39
6.9 Agent Manager log 39
6.10 Agent Manager log 40
6.11 Agent Manager log 40
6.12 Wireshark 42
6.13 Test results 44

List of Abbreviations

ACL	Access Control List.
API	Application Programming Interface.
CLI	command Line Interface.
DoS	Denial-of-Service.
GUI	Graphical User Interface.
LAN	Local Area Network.
MiTM	Man-in-The-Middle.
OF	OpenFlow.
OFPT	OpenFlow-based Parallel Transport.
ONF	Open Networking Foundation.
ONOS	Open Network Operating System.
SDN	Software-defined Networking.
TCP	Transmission Control Protocol.
TLS	Transport Layer Security.

1 Introduction

The aim of this paper is to perform a vulnerability analysis of an SDN controller. The focus is on attacks and tests that are initiated by a SDN switch to expose vulnerabilities of the SDN controller. In order to fulfill this aim, a fake switch was needed that primarily behaves like a legitimate switch, but then starts to perform tests to verify the security of the SDN controller. This led to another goal of this paper, which is the analysis of how it is most possible to imitate a switch using frameworks. For this purpose, a hybrid approach was chosen, consisting of a self-written program that imitates a switch and an existing framework called DELTA. The suitability of this approach and that of the individual components was tested and analyzed in the scope of this project. In addition, it was determined which attacks starting from the switch to the SDN controller are suitable, for which the possibilities of the hybrid approach were also examined. This resulted in several test cases, which were first applied to a SDN controller in a virtual test environment and then to an SDN controller located on the tabletop setup of the CORE project group. Based on the results of the test cases of the hybrid approach on different SDN controllers, an analysis was performed to identify the vulnerabilities of the respective SDN controller.

1.1 Structure

The basics of an SDN network, such as the SDN controller, SDN switch and the communication between them using OpenFlow, are described in Chapter 2. Chapter 3 continues with an evaluation of how a switch can be imitated and which approach will be used. In Chapter 4, the test environment is considered in more detail. The test cases are described in Chapter 5, and Chapter 6 evaluates the results of the test cases.

Chapter 7 summarizes the results and gives an insight on how these results could be used in further projects.

2 SDN Networking

Software-defined Networking (SDN) separates the control plane and the data plane in communication networks. While the data level is kept decentralized, the control level is logically centralized in a controller and serves as an interface for configuration purposes [8]. The general architecture of the SDN network is shown in Figure 2.1. There are three layers: The infrastructure, the control and the application layer. The infrastructure layer which consists of network devices such as switches (detailed explanation see 2.1) and router. These devices are only responsible for forwarding data packets. The control layer is the SDN layer in which the SDN controller is located. The centralized and programmable SDN controller is responsible for making the actual decisions in the SDN network. SDN controllers are considered the brain of an OpenFlow network. Every configuration and forwarding rules are defined by applications running on top of a controller. More information regarding the SDN Controller will be provided in 2.1. The Application Layer in the SDN network is the highest level of the network protocol and includes the interfaces between applications and the SDN controller. This layer defines network policies and network functions that are used by applications to access and control the network [18] [4].

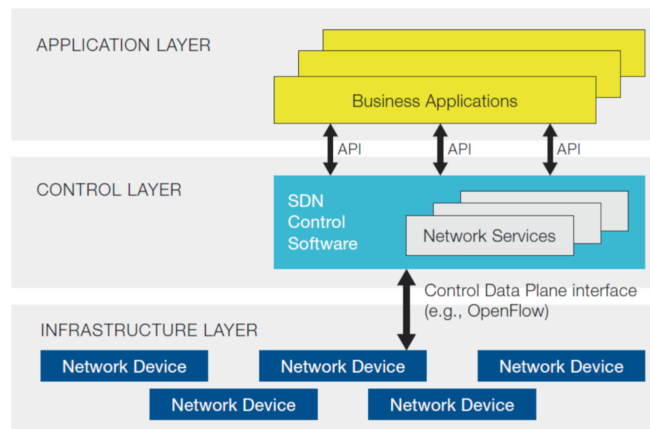

Figure 2.1: SDN Architecture [18]

The communication between the SDN controller and the switch, which reflects the separation of the control and data plane is defined by a particular protocol i.e., OpenFlow (see 2.3). Whereas the SDN enabled switch is often a dedicated networking device, the SDN controller is implemented as software and is usually deployed on a standard server component [8].

2.1 SDN Controller

The SDN Controller is a logically centralized software component in a Software-defined Networking (SDN) that automates network management and network control. The SDN controller acts as the core of the network operating system. The SDN controller is responsible for configuring network devices such as switches and routers to optimize traffic within the network. In general, the SDN controller enables network resources to be managed flexibly and applications to be deployed quickly by separating the network layer from the application layer. This allows the network to dynamically adapt to the needs of applications by defining policies that control traffic. The SDN controller acts as the core of the network operating system. It is responsible for managing network configurations and forwarding rules. It also manages multiple applications, provision resources, and enforce security or network policies. Meaning that changes only need to be made in one place, at the controller, and that individual devices no longer have to be changed when the network is modified. [26] [23]

There are several SDN controllers, including OpenDaylight [22], Floodlight [5], Ryu [2], and ONOS [25], all are open source software developed and supported by the community [26]. A detailed explanation of the controller frameworks is provided in 4.1.1.

2.2 SDN Switch

In general, a switch is a network device that is capable of controlling and forwarding data traffic within a Local Area Network (LAN). A switch is an important component of networks because it enables efficient use of network bandwidth and ensures reliable data transmission. A switch has multiple ports to which other network devices can be connected. The switch is able to analyze the data traffic and forward the data packets only to the port where the target device is connected, instead of forwarding to all connected devices. This means that data transmission is more efficient, as it is only sent to the devices that need it.

An SDN switch is a network switch used in the context of Software-defined Networking (SDN). In a SDN network, the switch is an important component of the data plane. In contrast to a traditional switch that makes decisions based on MAC addresses or other local routing tables, a SDN switch receives its instructions from the SDN controller, which automates network management and control as described above. Also unlike traditional switches that perform network control locally on the device, control for SDN switches is centralized and performed from the SDN controller. A SDN switch is a network device that is capable of forwarding traffic within the network based on the instructions it receives from the SDN controller. [12] [19]

2.3 OpenFlow Communication

OpenFlow is a protocol used in Software-defined Networking (SDN) to enable communication between the SDN controller and network devices such as SDN switches. OpenFlow was originally developed by a group of researchers at Stanford University and first published in 2008. Since 2011, OpenFlow has been maintained by the Open Networking Foundation (ONF).[18] A typical OpenFlow network would contain one

or more switches and one or more controllers. OpenFlow is a Layer 2 and 3 protocol with the basic objective of moving control of the network from individual network devices from different vendors to a centralized software controller. OpenFlow operates on top of TCP/IP. [18] [12] [11]

Communication between a SDN controller and a SDN switch is done using the OpenFlow protocol. OpenFlow is a standard protocol that serves as the basis for communication between the SDN controller and the SDN switch (see 2.2). An OpenFlow switch communicates with the controller through an OpenFlow channel. An OpenFlow channel can be encrypted by using TLS or run directly over TCP.

When a SDN switch receives a data request from a client, the packet is first forwarded to the SDN controller because the switch does not have local routing tables. The SDN controller then decides how to forward the packet, based on network policies set by administrators. Once the SDN controller makes a decision, it forwards the instructions to the SDN switch to forward the packet accordingly. The SDN switch is now able to forward traffic based on the SDN controller's instructions because of the flow table. A flow table is a table defined by OpenFlow that contains information about the data flows to be processed by the switch. Each entry in the flow table defines a rule that specifies how certain types of network traffic should be handled. An OpenFlow switch consists of one or more flow tables that perform packet search and packet forwarding. The controller controls the switch by adding, updating, and deleting flow entries via OpenFlow messages, proactively or reactively (in response to the arrival of new flows). The controller sends flow mod messages to the switches to update the flow tables on these devices. Each flow table entry consists of several fields, including a match field that defines what type of network traffic should be handled by the rule, and actions to be taken when a match is found. [12] [1] [8]

The communication between the SDN controller and the SDN switch is thus bidirectional. The SDN controller sends instructions to the switch, and the switch sends reports and statistics back to the controller. This enables the controller to monitor and optimize network performance in real time. This leads to three different kinds of messages: controller-to-switch, asynchronous and symmetric. Under the respective message category are the individual message types.

Controller-to-Switch Messages Controller-to-switch messages are initiated by the controller and used to directly manage or inspect the state of the switch. The following is an excerpt from the OpenFlow Switch Specification [7] to explain the possible message types:

- **Features** — The controller requests the basic capabilities of a switch by sending a features request. The switch must respond with a features reply that specifies the basic capabilities of the switch.
- **Get Config** — The controller sets and queries configuration parameters in the switch. The switch only responds to a query from the controller.
- **Modify-State** — The controller sends Modify-State messages to manage state on the switches. Their primary purpose is to add, delete, and modify flow or group entries in the OpenFlow tables and to set switch port properties.
- **Read-State** — The controller sends Read-State messages to collect various information from the switch, such as current configuration and statistics.
- **Packet-out** — These are used by the controller to send packets out of the specified port on the switch, or to forward packets received through packet-in messages. Packet-out messages must contain a full packet or a buffer ID representing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified. An empty action list drops the packet.
- **Barrier** — Barrier messages are used to confirm the completion of the previous operations. The controller sends a Barrier request. The switch must send a Barrier reply when all the previous operations are complete.
- **Role-Request** — Role-Request messages are used by the controller to set the role of its OpenFlow channel, or query that role. It is typically used when the switch connects to multiple controllers.

Asynchronous Messages Asynchronous messages are initiated by the switch and used to update the controller about network events and changes to the switch state. The following is an excerpt from the OpenFlow Switch Specification [7] to explain the possible message types:

- **Packet-In** – Transfer the control of a packet to the controller. For all packets forwarded to the Controller reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers. Other processing, such as TTL checking, can also generate packet-in events to send packets to the controller. The packet-in events can include the full packet or can be configured to buffer packets in the switch. If the packet-in event is configured to buffer packets, the packet-in events contain only some fraction of the packet header and a buffer ID. The controller processes the full packet or the combination of the packet header and the buffer ID. Then, the controller sends a packet-out message to direct the switch to process the packet.

- **Flow-Removed** – Inform the controller about the removal of a flow entry from a flow table. These are generated due to a controller flow delete request or the switch flow expiry process when one of the flow timeouts is exceeded.

- **Port-status** – Inform the controller of a state or setting change on a port.

- **Error** – Inform the controller of a problem or error.

Symmetric Messages Symmetric messages are initiated by either the switch or the controller and sent without solicitation. The following is an excerpt from the OpenFlow Switch Specification [7] to explain the possible message types:

- **Hello** – Hello messages are exchanged between the switch and controller upon connection startup.

- **Echo** – *Echo request* messages can be sent from either the switch or the controller and must return an *echo reply*. They are mainly used to verify the liveness of a controller-switch connection.

[7], [10], [16]

Communication Procedure Here is a typical sequence of messages exchanged when establishing a connection between an SDN controller and a switch:

1. The SDN controller sends a Hello message to the switch to establish a connection.
2. The switch responds with a Hello message to confirm the connection.
3. The SDN controller sends a Features Request message to the switch to retrieve information about the switch's capabilities and features.
4. The switch responds with a Features Reply message containing the requested information.
5. The SDN controller sends a "Configuration Request" message to the switch to change the switch's configuration.
6. The switch responds with a Configuration Reply message to confirm the success or failure of the configuration change.

These steps can vary depending on the use case, and other messages can be exchanged to control and program the flow of data in the network. But in general, this is the basic flow of communication between an SDN controller and a switch using OpenFlow messages. [17] [16] [8] [1]

In this document, in addition to the abbreviation OF, which denotes the OpenFlow standard, the abbreviation OFPT will also be used or seen in the code snippets. OFPT is used to describe the different types of OpenFlow messages.

3 Possibilities of Impersonating a Switch

The goal of this project is to imitate a switch using software, successfully connect to the SDN controller, exchange OpenFlow messages with it, and perform attacks on the controller in addition to imitating normal operation of the switch. In order to fulfill this goal, some decisions had to be made in advance on how to implement this.

This chapter describes the possibilities to create a virtual switch that can communicate with an OpenFlow controller. The various options were considered from the perspective that the goal is to have a virtual software-based switch that can successfully connect to the SDN controller and then be able to perform attacks on the SDN controller through extensions to normal operation. For this purpose, Section 3.1 examines which frameworks exist that are already adapted to the use case in which attacks are to be carried out from the switch on the SDN controller. The focus here is primarily on the DELTA framework. In Section 3.2, we examined the possibilities of implementing a virtual switch without a framework. The focus here is on explaining why the effort of a self-written program can be justified and what advantages it offers over other solutions. Furthermore, this section discusses which technical decisions had to be made for the program to meet the requirements. After many decisions have been made in the two sections, Section 3.3 will summarize them and explain in what way which solution approach is suitable to perform the test cases (attacks). This decision forms the foundation for the further process.

3.1 Existing Frameworks - DELTA

The first step was to conduct a research for existing solutions for penetration testing in SDN networks regarding the OpenFlow communication between SDN controller and switch. During this research, it was discovered that part of the CORE project group has already begun to investigate such a framework: The DELTA: SDN Security Evaluation Framework (hereafter referred to as DELTA). DELTA is a framework for evaluating the security of SDN systems. The framework provides a set of tools and methods to identify and analyze potential security vulnerabilities in SDN systems. It supports various security analyses such as penetration testing, fuzzing testing, and vulnerability assessments. The DELTA framework aims to provide SDN system developers and security professionals with a tool to improve the security of SDN systems and minimize potential attack vectors [20], [15], [3]. Since the extensive setup had already been carried out by the CORE project group and was therefore immediately ready for use, it was a reasonable step to abandon the previous research and focus on DELTA. DELTA also offers, among other things, a variety of test cases that perform attack scenarios using OpenFlow messages sent from the switch to the SDN controller. It therefore meets the basic requirements for this project. The architecture of DELTA and its components is shown in Figure 3.1.

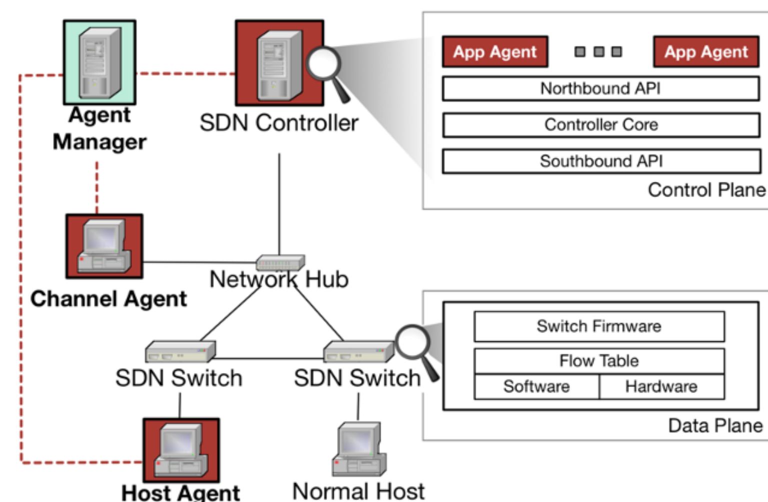

Figure 3.1: DELTA Architecture [3]

DELTA consist of four different agents which are described as follows on GitHub [3]:

- **Agent Manager** — It controls the other agents which are deployed on the target SDN network. It also analyzes the test results collected from the agents.
- **Application Agent** — It conducts attack procedures as instructed by the manager. It implements the known malicious functions as an application agent library.
- **Channel Agent** — It is deployed between the control plane (controller) and the data plane (switch). The agent sniffs and modifies the unencrypted control messages.
- **Host Agent** — It behaves as if it were a legitimate host participating in the target SDN network. It generates network traffic as instructed by the agent manager (e.g. DDoS, LLDP injection, etc).

DELTA supports the most common controllers like Floodlight [5], ONOS [25], OpenDaylight [22], and Ryu [2] and any OpenFlow enabled switches (including software switches).

There are three main test sets within DELTA:

- Test set 1 refers to the data plane security, therefore on the OpenFlow messages from a controller to a switch.
- Test set 2 refers to the control plane security, therefore on the OpenFlow messages from a switch to a controller.
- Test set 3 refers to advanced security, those are sophisticated security tests exploiting a variety of vulnerabilities (e.g. SDN applications exploiting SDN controllers' architectural vulnerabilities).

[14] [15]

3.2 Self-Written Program

Considerations of a Self-Written Program to Simulate a Switch Several options for implementing a switch for the test scenario were considered. Requirements for this were that the switch be maximally modifiable so that various attacks could be carried out. For this purpose, the code must be constantly adapted and extended in order to be able to deal with different attack scenarios. Options like OpenVSwitch [6], which is an open-source software that serves as a virtual switch for managing network connections in virtualization environments, were considered. However, they were too complex for our application purpose and not specifically customized for penetration testing, which is why virtual switch options were discarded as a possible option. The general advantages of writing your own program to simulate a switch are, among others, the complete control over the behavior and functionality of the switch and the possibility to customize or extend functions and protocols. There is no need to deal with other software and discuss what exactly is needed and what is not, and there is no need to check in advance whether the functions are sufficient to achieve the goals. However, coding a custom switch also requires more development time, but still offers the advantage of implementing everything you want and being able to focus exactly on the desired use cases - in this case, attack scenarios. In addition to these considerations, the biggest advantage of a custom program was that a maximum number of options must be available, especially for attack scenarios. It must be possible to change the program without restriction so that an attack can be as successful as possible from the attacker's point of view. Static solutions that make only one path possible are too limiting for this. For this reason, it offered itself to write an own program that simulates a switch. So, it could be implemented that only necessary properties are contained in it and the structure is clear and easy to handle. In addition, an own program allows maximum freedom in the design of the attack vectors.

OpenFlow Library To imitate a switch, it must be clarified how it communicates with the SDN controller. This communication takes place via the OpenFlow protocol already introduced in 2.3. The first step was to research which OpenFlow library was the most suitable to create the OpenFlow messages in the program. The requirements

for a suitable OpenFlow library are very simple. First, it must be well documented how the connection establishment can be implemented using the messages. Then the essential point is that it must be possible to create all existing OpenFlow messages independent of the OpenFlow version. All OpenFlow libraries offer this. So, it is important in the next step to see on which language the library should be based, because the program code must be programmed in the same programming language. Here the decision was made that Python would be suitable since Python programming language is typically used for programming OpenFlow messages, as it provides a simple and effective way to implement and test network protocols. Thus, the selection of possible libraries was limited to Python. Finally, the comprehensiveness and the type of documentation as well as the uncomplicated handling were decisive. It was important that every single OpenFlow message and its fields are documented in detail and that it is possible to start using the library immediately. These requirements were satisfactorily met by PyOF (Python OpenFlow) from Kytos [13].

PyOF is a Python library that allows to create, process, and analyze OpenFlow messages. It is part of the Kytos project, which provides an open-source software platform for network automation. PyOF is an important part of Kytos because it provides a powerful API for programming network devices that support the OpenFlow standard. PyOF provides an API that allows you to create and receive OpenFlow messages to communicate with another OpenFlow device. Thus, by using PyOF, a virtual switch can be created that can communicate with an OpenFlow controller. According to this, it is software-based possible to imitate a switch with PyOF. [13] Thus, the most basic requirements for the program were met. The detailed program structure is described in section 4.1.2.

3.3 Hybrid Solution

On the one hand, the DELTA framework was considered, and it was discussed that its functionality and test cases are useful and target oriented. On the other hand, we evaluated how to make the test cases even more flexible and extensible than is possible in DELTA. For this it was decided that a self-written program to simulate a switch is more target-oriented and most flexible. Since both approaches have their advantages

and disadvantages, it makes sense not to commit to one solution approach but to choose a hybrid solution. For this reason, both DELTA tests and self-written tests will be used to test the security of the switch controller communication. These tests are defined and described in more detail in Chapter 5.

4 Test Setup

This chapter forms the basis of how the test environment looks in detail. In the previous chapters, the basics about the components of the SDN network and the possibilities of faking a switch were explained. In addition, an initial decision on how the attacks are to be carried out has already been made in Chapter 3.3.

In this chapter, these individual fundamentals will now be concretized. The goal is to obtain a detailed overview of the test environment and the individual components involved in the test cases. For this in 4.1, the components are introduced which are necessary for the project. In 4.2, the entire test environments are examined in more detail, i.e., the locations where the attacks will be carried out and in which form the components presented in 4.1 will be used in each case.

4.1 Test Components

In the following, the most important components of this project will be discussed. In 4.1.1, it is specified that the ONOS framework will be used for the SDN controller. In 4.1.2, the structure of the custom program to implement a pretend switch is described. And in 4.1.3 it is described how DELTA was integrated into this project.

4.1.1 ONOS Controller

For the SDN Controller was a controller framework required. Several controller frameworks exist, such as such as Floodlight [5], Ryu [2], the Open Network Operating System (ONOS) [25], and OpenDaylight [22]. For this project the decision was made based on the controller framework which would be used when testing the test cases on the tabletop setup. The Open Network Operating System Framework is running there

which will be referred to as ONOS in the following. That is why the decision was made for ONOS. For this reason, a comprehensive analysis and comparison of the other frameworks was not carried out. ONOS is an open-source controller framework for Software-defined Networking (SDN) that provides a powerful API, tools for monitoring and analyzing networks, and a modular architecture. ONOS is known for its scalability as supported by the Open Networking Foundation (ONF) and by an active community of developers. ONOS provides the control plane for a software defined network. The ONOS platform and applications act as an extensible, modular, distributed SDN controller. ONOS therefore enables user to manage network components such as switches and links. ONOS applications and use cases often consist of customized communication routing, management, or monitoring services for software-defined networks. Thus, it also fulfills the prerequisite needed for this test [25].

Detailed information about the respective ONOS version and the additionally installed app services can be found in Chapter 4.2.

4.1.2 Self-Written Program

The entire program can be inspected in Appendix 1 (Impersonated Switch.py).

Integration of the Python OpenFlow Library from Kytos As already mentioned in 3.2, the Python SDN Library from Kytos is used to create SDN messages. The library is built in a way that the SDN messages can be imported depending on the version. For this project the SDN version 04 was used. The message types are also divided into symmetric, asynchronous and controller2switch messages as described in 2.3. In addition, the library also contains common elements in which further tools are provided to better access the message header or to unpack the SDN message to better access individual items by code. In the code snippet (see Figure 4.1) you can see some examples of imports.

```
26  from pyof.v0x04.common.utils import unpack_message
27  from pyof.v0x04.common.header import (Type, Header)
28  from pyof.v0x04.symmetric.hello import (Hello, HelloElemHeader, HelloElemType, ListOfHelloElements)
```

Figure 4.1: Example Imports

Connection Establishment and First Hello Before the actual SDN protocol connection between SDN controller and switch can be established, a transport connection must be established as the first stage. This can be either a TCP or a TLS connection. In this case it is a TCP connection between SDN controller and switch. The code snippet (see Figure 4.2) shows the implementation of the TCP connection setup between switch and SDN controller.

```
def set_connection(tcp_ip, tcp_port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(("127.0.0.1", 34564))
    s.connect((tcp_ip, tcp_port))
    return s

def main():
    """Init"""
    buffer_size = 16  # In bytes
    tcp_socket = set_connection('10.0.2.15', 6653)
    start_communication(tcp_socket, buffer_size)  # 1st Hello by Switch
    buffer_size = 8

    processing_queue = Queue()

    t1 = threading.Thread(name='receive_thread', target=receive_thread, args=(processing_queue,tcp_socket, buffer_size,))
    t2 = threading.Thread(name='reply_thread', target=reply_thread, args=(processing_queue, tcp_socket,))

    t1.start()
    t2.start()

    while 1:
        time.sleep(1)
```

Figure 4.2: Main with set connection

As soon as a successful connection has been established, the switch starts the communication with the *start_communication(tcp_socket, buffer_size)* function. The first Hello message is sent from the switch to the SDN controller. The code snippet in Figure 4.3 shows this process.

```
def _new_list_of_hello_elements():
    hello_elem = HelloElemHeader(HelloElemType.OFPHET_VERSIONBITMAP, length=8, content=b'\x00\x00\x00\x10')
    elements = ListOfHelloElements()
    elements.append(hello_elem)
    return elements


def start_communication(tcp_socket, buffer_size):
    """Send and receive initial Hello Message"""
    send_message(tcp_socket, Hello(62, _new_list_of_hello_elements()))
    data = tcp_socket.recv(buffer_size)
    print("received data: ", data)  # Print received raw data
    tmp = unpack_message(data)
    print("Received type: ", tmp.header.message_type)
```

Figure 4.3: Start communication

First, a Hello message is created using the PyOF library in the function *new_list_of_hello_elements()*. The PyOF library also provides a variety of examples of how a particular message type can usually be filled with information, which has made building OF messages for this project much easier overall. In general, and also in this case, the OF messages are created using the *send_message(tcp_socket, msg)* function (see code snippet in Figure 4.4) to the SDN controller.

```
55  def send_message(tcp_socket, msg):
56      print(msg)
57      try:
58          tcp_socket.send(msg.pack())
59          print("send data:", msg.pack())
60      except:
61          print("TCP Error send")
```

Figure 4.4: Send message

After sending the Hello message, the switch actively waits once for the response of the SDN controller to ensure that a Hello message comes back and that the initial Hello handshake for establishing the connection was successful.

Regular Operating After these prerequisites were met, consideration was given to how the flow of the program could best be optimized. Since the simulated switch must behave like a real switch, it must be ensured that it responds quickly and efficiently to messages from the SDN controller, i.e. that it can simulate and fulfill the normal operation of a switch well. To implement normal operation as efficiently as possible, a thread-based approach was chosen. This approach is shown in Figure 4.5.

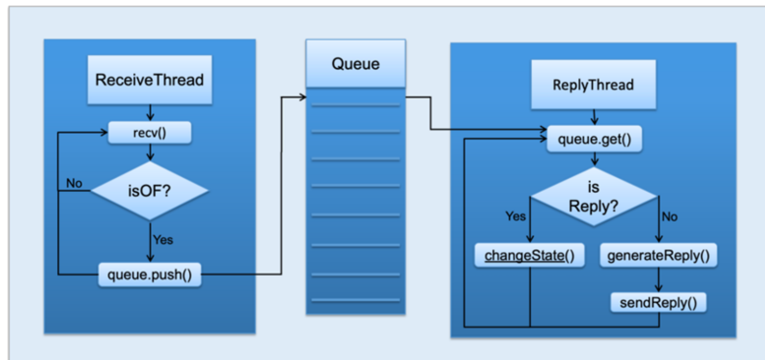

Figure 4.5: Thread-based approach

There are two threads, the Receive and the Reply thread. The Receive Thread is responsible for checking if the incoming message is an SDN message, if so it will be processed and pushed into the message queue. The Reply Thread fetches the first SDN message from the queue at the beginning. Two types of messages are distinguished. The first type are the answers of the controller, these lead to the fact that the controller does not expect an answer, but expects from the switch to accomplish internal changes as for example to delete a flow entry in the flow table of the switch. The other message type is status queries from the SDN controller where the controller expects a suitable response from the switch. In the code snippet (see Figure 4.6) you can see the structure of the receive and reply thread in the code.

```
224 def reply_thread(queue, tcp_socket):
225     print(threading.currentThread().getName(), 'Starting')
226     while 1:
227         try:
228             data = queue.get()
229             received_msg = unpack_message(data)
230             print("QUEUE  GET", received_msg)
231             handle_request_messages(tcp_socket, received_msg, data)
232         except queue.Empty():
233             pass
234         except:
235             print("ERROR Reply Thread")
236
237
238 def receive_thread(queue, tcp_socket, buffer_size):
239     print(threading.currentThread().getName(), 'Starting')
240     while 1:
241         try:
242             data = tcp_socket.recv(buffer_size)
243             if data != b'':
244                 length = int.from_bytes(data[2:4], "big")
245                 if length >= 8:
246                     data_body = tcp_socket.recv(length - 8)
247                     data = b"".join([data, data_body])
248
249                     print("QUEUE  PUT", data)
250                     queue.put(data)
251                 print("")
252         except:
253             print("ERROR Receive Thread")
```

Figure 4.6: Reply thread

The messages are taken from the byte stream when they are received and placed in the queue. The reply thread takes the message, unpacks it so that it can be worked with better in the further course and calls the function *handle_request_messages(tcp_socket, received_msg, data)*. In this function first the message type is extracted from the header as well as the xid of the request, which serves as ID of the communication and must

be contained in the answer. After that a query is made which message type it is. Once the message type has been identified, the reply message is prepared using PyOF and sent using the *send_message(tcp_socket, msg)* function. Messages that do not require a response from the switch will not be discussed further. The handle request method is the core of the application. From here we simulate the normal operation of the switch and respond to appropriate requests from the SDN controller to maintain the trusted connection. In the code snippet (see Figure 4.7) you can see a part of the handle request function.

```
65  def handle_request_messages(tcp_socket, received_msg, data):
66      msg_type = received_msg.header.message_type
67      req_xid = received_msg.header.xid
68      if msg_type == Type.OFPT_HELLO:  # 0 -> 0
69          msg = Hello(req_xid, _new_list_of_hello_elements())
70          send_message(tcp_socket, msg)
71      elif msg_type == Type.OFPT_ECHO_REQUEST:  # 2 -> 3
72          msg = EchoReply(req_xid)
73          send_message(tcp_socket, msg)
74      elif msg_type == Type.OFPT_FEATURES_REQUEST:  # 5 -> 6
75          msg = FeaturesReply(req_xid, DPID('00:00:00:00:00:00:00:01'), 0, 254, 0, 0x0000004f, 0x00000000)
76          send_message(tcp_socket, msg)
77      elif msg_type == Type.OFPT_GET_CONFIG_REQUEST:  # 7 -> 8 (-> 9 C)
78          msg = GetConfigReply(req_xid, ConfigFlag.OFPC_FRAG_NORMAL, ControllerMaxLen.OFPCML_NO_BUFFER)
79          send_message(tcp_socket, msg)
```

Figure 4.7: Handle request

4.1.3 DELTA

At the beginning, it had to be ensured that DELTA is also compatible with the SDN controller. It turned out that DELTA is compatible with ONOS, so this solution is compatible with the test setup. As already mentioned in Section 3.1 the DELTA framework was already discovered and prepared by the CORE project group. Thus, it was at the stage that DELTA was ready for deployment and could be used for the research purposes of this work. Accordingly, no further preparation was necessary here, but this section briefly describes how DELTA can be deployed in general, so that one can better understand the test setup.

The DELTA project offers a virtual machine image file specifically designed for quick and easy setup of DELTA in a virtual environment. This approach has already been chosen by the project group and set up on both the table setup and for personal use in a virtual environment.

The virtual machine image contains a preconfigured version of Ubuntu Linux and of all the necessary dependencies and tools to run DELTA. It also contains a Graphical User Interface (GUI) for using DELTA. The structure of the DELTA test environment within the virtual machine is shown in Figure 4.8. The individual components, i.e. the agents, have already been introduced in Section 3.1.

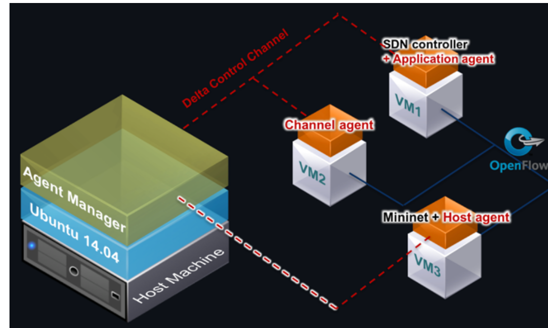

Figure 4.8: DELTA test environment

4.2 Test Environment

There are two environments in which the tests have been performed. The self-written program and the test cases contained therein, and the tests of the DELTA framework were initially performed in a virtual test environment which consists only of virtually generated entities. The test environment acts as a preliminary stage for testing the program and framework on the tabletop setup. In the following, both are analyzed in more detail. This analysis serves, among other things, as a basis for the analysis of the resulting test results and, if applicable, the differences contained therein.

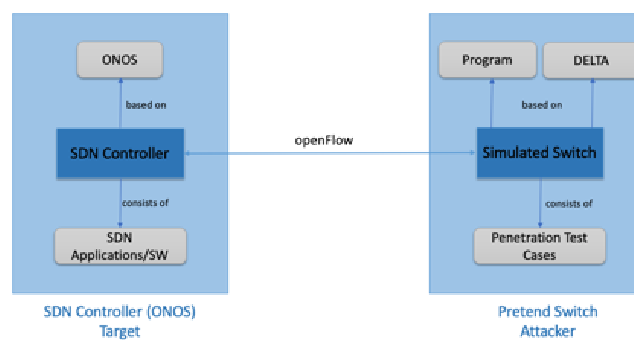

Figure 4.9: Test setup structure

The general structure of the test setup is shown in the diagram above (see Figure 4.9). This general overview is applicable for both test environments. The entities have been explained before in 4.1. The test environments consist of a SDN controller based on ONOS (see Section 4.1.1). For the tabletop setup ONOS is installed in the version Sparrow (2.2.2) [21] as a service, which is automatically started with the standard apps of the table setup during system startup. For the virtual test environment, the ONOS version Tucan (2.3.0) [21] is used as this was the latest ONOS version while performing the tests. To configure ONOS for the test scenario, additional app services had to be installed, which were also used for the table setup. The following SecVi table setup app services are installed for both environments:

- **StaticForwarding App** - Must be enabled, implements static rules of the table structure.
- **ReactiveForwarding App** - Must be enabled, implements dynamic forwarding using an Access Control List (ACL). For this, the ACL entries are installed from a stored file. The rules are displayed in the GUI. Installed flows and violations are also reflected in a GUI. [9]

Only these and the ONOS default settings were used for the virtual test environment, since the SDN controller does not have a larger purpose and is only used to interact with test switches and the pretended malicious switch. For the tabletop setup, however, additional applications are installed, as this SDN controller must perform several functionalities, since this is the setup of a vehicle network. These additional applications like anomaly detection, ONOS apps and optional apps do not play any further role here in detail for now.

The SDN controller communicates with a variable number of switches via the OpenFlow protocol. For the tabletop setup, there is an additional number of devices connected to two switched, since this is an active software defined network and the controller thus configures and manages additional devices. In the virtual environment, on the other hand there will be at the beginning no other switches linked to the controller. The first aim will be to add an impersonated switch to the network. In further test cases authentic switches will be connected to the SDN controller before

the fake switch will be added. OpenFlow version 1.3 (v0x04, or OF1.3) is used for both test environments.

The following diagram (see Figure 4.10) shows the topology of the table structure. Either the program and the DELTA framework are located on the core-NUC2. The SDN controller is set up on the core-NUC1.

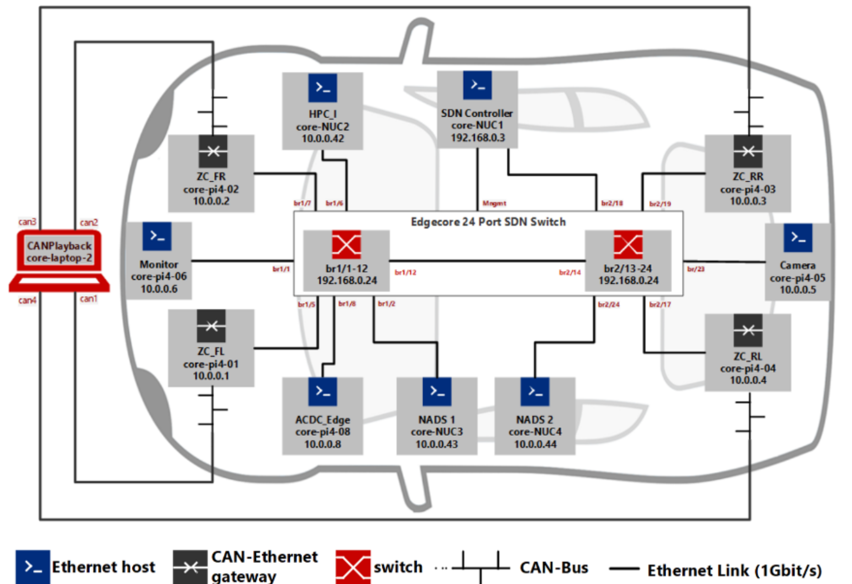

Figure 4.10: Table structure topology

The simulated switch, therefore, the fake switch is based on the self-written program (see 4.1.2) and on the DELTA framework (see 4.1.3) which were mentioned before. Both approaches focus on connecting to the controller using OpenFlow protocol and becoming a legitimate part of the SDN network from the controller's perspective. Once this has been achieved, both approaches have various test cases for carrying out further penetration tests. At this point, there is no difference between the virtual test environment and the table setup.

5 Test Cases

This chapter describes the test cases to be performed both in the virtual environment and on the tabletop setup. The test environments have already been described in detail in Section 4.2. Test Cases executed with both the DELTA framework (see 3.1 and 4.1.3) and the self-written program (see 3.2 and 4.1.2). The main focus will be on DELTA because the framework brings a lot of relevant tests. However, during the deployment it was noticed that not all tests are executable as required, so the already self-written program was added as an addition to these and other test cases. The DELTA test cases are described in Section 5.1 and the self-written program in Section 5.2.

5.1 DELTA Attacks

As already discussed in Section 3.1 there are three main test sets within the DELTA framework. The test set number 2 refers to the control plane security which means that it targets the controller by attacks with focus on the OpenFlow messages from the switch to the controller. Therefore, test set 2 is particularly suitable for performing the tests. In the following, the planned test cases are considered, and the respective attack is described. The test cases are assigned a number. This number is the identification of the test, which will be used throughout this document to identify the respective test.

2.1.010 – Malformed version number

The first test which will be performed is test case number *2.1.010 – Malformed Version Number*. The version number in an OpenFlow message between a SDN controller and a switch is used to identify the version of the OpenFlow protocol used by the two endpoints. The OpenFlow protocol is updated periodically to add new features, fix

bugs, or improve performance. The version number is an 8-bit field in the OpenFlow header. The header is the first part of an OpenFlow message and contains information such as the version number, the length of the message, the type of the message, and the ID of the transaction (xid) (see Figure 5.1).

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version;    /* OFP_VERSION. */
    uint8_t type;       /* One of the OFPT_ constants. */
    uint16_t length;    /* Length including this ofp_header. */
    uint32_t xid;       /* Transaction id associated with this packet.
                           Replies use the same id as was in the request
                           to facilitate pairing. */
};
```

Figure 5.1: OpenFlow header [7]

The version number allows endpoints to determine which functions and features of the protocol are supported and how messages should be interpreted. By specifying the version number, the SDN controller and switch can ensure that they are using the same protocol and that messages can be exchanged effectively. If the version numbers do not match, this can cause incompatibilities and affect the proper operation of the network.

In summary the *Malformed Version Number* test is a method to test the communication between SDN controller and switch for compatibility problems. In this process, an incorrect version number is set in the OpenFlow message header by the switch and sent to the SDN controller. If the SDN controller does not recognize the incorrect version number and does not respond appropriately to the message, it means that this attack was successful and that a potential exploitable vulnerability has been discovered.

2.1.020 – Corrupted Control Message (Corrupted Content)

In the *Corrupted Control Message* test case, the switch simulates an error or corruption in the OpenFlow message by changing either the header or the content of the message. The switch then sends the corrupted message to the controller and expects a response.

When the controller receives a corrupted message, it should, at best, check whether the header or the contents of the message are valid. If it is not, the controller will send an error message to the switch stating that the message is corrupted and cannot be processed. This test checks if the controller is able to detect corrupted OpenFlow messages and react appropriately to them. The idea behind this test is to check the resistance of the controller against potential attacks on the network. By sending corrupt messages, an attacker can try to make the controller perform unexpected actions or compromise the network. The test is intended to ensure that the controller is capable of detecting and defending against such attacks.

2.1.030 – Handshake without Hello

In the *Handshake without Hello* test, an OpenFlow connection request is sent by the switch to the SDN controller without a "Hello" message having been sent beforehand. A "Hello" message is a special OpenFlow message that is used to establish the connection between the switch and the SDN controller and to exchange the supported version of the OpenFlow protocol (more information regarding the handshake see Section 2.3). Figure 5.2 shows a correct and complete Hello Handshake.

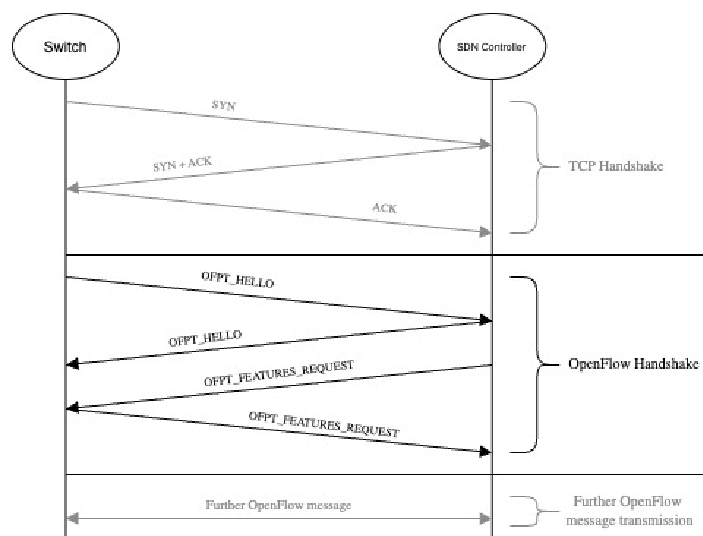

Figure 5.2: Complete Hello handshake [28], [29], [10]

If the SDN controller accepts the connection without receiving a "Hello" message and sends a response message, this indicates that this attack was successful and that there is a potential vulnerability in this scenario. A properly implemented SDN controller should not send a response without first performing a full handshake to establish the connection. The SDN controller should terminate the connection if no "Hello" message has been received.

2.1.040 – Control Message before Hello

In the *Control Message before Hello* test case, the OpenFlow Hello handshake (see Figure 5.2 above for the correct and complete procedure) is not performed at all. The switch starts the communication by directly sending a packet-in OpenFlow message. This test is used to evaluate if the SDN controller has sufficient protection against control communication prior completed connection establishment. The SDN controller has two possibilities to react to this control message, in this case a packet-in, sent by the switch.

If the SDN controller is sufficiently secured, the expectation is that it will simply ignore the message from the switch and neither respond nor trigger any resulting actions. However, if it should respond to this message or even trigger actions, a major vulnerability is revealed, since at this point, without a completely correctly executed handshake, a switch can simply send control messages to the SDN controller. If the test result is FAIL, i.e. the controller is vulnerable to this attack. This test can be extended as required with messages other than packet-in. The self-written program presented in 3.2 and 4.1.2 is suitable for such test extensions.

2.1.050 – Multiple Main Connection Request from Same Switch

This *Multiple Main Connection Request from same Switch* test checks if a switch attempts to connect multiple times as the main connection to a SDN controller. In such a scenario, the switch would attempt to establish multiple connections to the controller, which

may cause the network connection to be disrupted. This can happen if the switch sends the "Connect" OpenFlow protocol message multiple times without first disconnecting from the controller. This can happen for a variety of reasons, such as a disconnection or network malfunction. The test checks whether the SDN controller is able to detect and prevent such situations by ensuring that only one connection is set up as the main connection. The test runs in such a way that the dummy switch performs a complete and successful handshake with the controller. This handshake is then attempted again from the switch. If the second handshake is executed, the result of this test is that the controller has a potential vulnerability. However, if the second handshake is rejected, the controller behaves as required and passes the test.

5.2 Self-Written Program

After it was possible to successfully pass the first test case with the self-written program, namely to establish the connection with the SDN controller, and the execution of further tests was planned, attention was drawn to the DELTA framework. Since the DELTA framework performs a large number of tests that were intended for the self-written program or had already been started to be implemented, there was no need to implement them there any further. For example, the Malformed Version Number test implemented and contained in the program is no longer listed here for this reason. However, since in practice concerning DELTA a few tests had not worked in all test environments or in general, for this the self-written program was expanded by these tests. For this reason, the initial idea of replacing the program with DELTA was abandoned. Instead, the decision was made for a hybrid solution. Both approaches to test the SDN controller are equally strong and useful. The tests that can be implemented well by DELTA have already been described in detail in Section 5.1. Tests that can only be partially implemented in DELTA, or not at all, such as test case number *3.1.010 Packet-in Flooding*, were therefore implemented in this program. The reason for leaving the program as an active part of this project is that precisely such cases, in which static frameworks reach their limits, can be solved in a flexible way with the program. Likewise, the test cases can be extended as needed. This is hardly possible within the DELTA framework. The test cases for the self-written program can be recognized by

the "#" which stands in front of the test case number. In the Section 4.1.2 the structure and process of the program was already described in detail. It is still to be clarified at which point in the code the tests should be carried out and above all also when an attack is to be inserted into the process. Depending on the attack, it makes sense to first wait for the handshake and only start the attack when normal processing is ongoing (of course, this does not apply to attacks such as handshake without hello etc.). This normal processing takes place in the handle request function of the program. Since the SDN controller continues to communicate with the switch after the initial settings have been shared, it makes sense to take a close look at the message history before launching the attack. For this project, the message history was analyzed, and the attacks were initiated based on this analysis when certain messages were received.

#0 - Initial Test

The initial test and precondition is the general successful establishment of a connection between the simulated switch and the SDN controller, as well as the exchange of fake messages from the switch. This may seem like a rather irrelevant simple test in the first place, but if the SDN controller performs the connection setup and sees the simulated switch as part of the network, the SDN controller has an incorrect view of the network topology. So, it is a first step to be able to influence the perception and behavior of the SDN controller and possibly offers further possibilities for malicious attacks. A successful connection means that the SDN controller trusts the pretend switch and is potentially unable to distinguish between a legitimate switch and a pretend switch. This could potentially lead to attacks, as the pretend switch may be able to manipulate the controller and perform unwanted actions. The code snippet of the handshake was already shown and described in 4.1.2.

#2.1.020 – Corrupted Control Message (Corrupted Content)

This test is performed in addition to the DELTA test, as the DELTA test proved not to be completely sufficient, or the result was not comprehensible enough. For this reason, the decision was made to add this test to the program. In the Corrupted control message test case, the switch simulates an error or corruption in the OpenFlow message by changing either the header or the content of the message. The switch then sends the corrupted message to the controller and expects a response. A more detailed explanation can be found in 5.1.

The test is divided into two parts:

1. ***Corrupted Control Message (Corrupted Content)* during connection setup:** Here the corrupted message is sent during the initial handshake. This test is used to check whether a new switch can already be added to the network in this way, despite corrupted messages.
2. ***Corrupted Control Message (Corrupted Content)* during normal operation:** In this test scenario the corrupted message is sent after the handshake, after the switch has been communicating with the SDN controller for a while. The corrupted content will be inserted in a required response for the SDN controller to make sure that he processes it. This test aims to check if it makes a difference if an existing switch is compromised instead of a new switch. So whether the SDN controller treats corrupted messages from already successfully connected switches differently.

#3.1.010 – Packet-in Flooding

Packet-in flooding is a test in which the switch intentionally sends a large number of packet-in messages to the SDN controller to test its responsiveness and performance. This tests whether the controller is able to process the large number of packet-in

messages and respond quickly enough to effectively control the switch. This test is important to ensure the scalability and robustness of the SDN network. The undesirable behavior of the SDN controller during a packet-in test would be if it was unable to effectively process or respond appropriately to all packet-in messages. Here are some possible undesirable behaviors of the controller:

- Overload: If the controller is not able to handle the high number of packet-in messages, this can lead to an overload of the controller, which can cause performance problems in the network.
- Delays: If the controller does not respond quickly enough to the packet-in messages, this can cause delays in the network, resulting in poor performance and user experience.
- Incorrect decisions: If the controller makes incorrect decisions or performs incorrect actions due to overload or delays, this can cause errors in the network that can lead to packet loss, network disruptions, or security problems.

Overall, inadequate, or ineffective behavior of the controller during a packet-in-test can lead to poor performance, scalability, and reliability of the SDN network [24].

This test is part of the DELTA Test set 3 - Advanced Tests (more about this in 3.1). However, the execution of this test was not possible, therefore this test has been included in the program, allowing this test to be performed.

The decision was made to start packet-in flooding after a *Packet_Out* message is received. For this the *send_packet_in(req_xid, tcp_socket)* function is executed (see Figure 5.3). In this function, *Packet_In* messages with all their required data contents are created and sent to the SDN controller in a loop without interruption. Further normal operation of the program is intentionally interrupted for this purpose.

```
211  def send_packet_in(req_xid, tcp_socket):
212      oxmtlv = OxmTLV(oxm_class=OxmClass.OFPXMC_OPENFLOW_BASIC,
213                      oxm_field=OxmOfbMatchField.OFPXMT_OFB_IN_PORT,
214                      oxm_hasmask=False, oxm_value=b'\x00\x00\x00\x02')
215      match = Match(match_type=MatchType.OFPMT_OXM, oxm_match_fields=[oxmtlv])
216      data_in = b'\x33\x33\x00\x00\x00\x16\x92\xfd\x3d\x2a\x06\x0c\x86\xdd\x60\x00'
217      data_in += b'\x00\x00\x00\x24\x00\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
218      data_in += b'\x00\x00\x00\x00\x00\x00\xff\x02\x00\x00\x00\x00\x00\x00\x00\x00'
219      data_in += b'\x00\x00\x00\x00\x00\x16\x3a\x00\x05\x02\x00\x00\x01\x00\x8f\x00'
220      data_in += b'\x69\x54\x00\x00\x00\x01\x04\x00\x00\x00\xff\x02\x00\x00\x00\x00'
221      data_in += b'\x00\x00\x00\x00\x00\x01\xff\x2a\x06\x0c'
222      while 1:
223          msg = PacketIn(req_xid, OFP_NO_BUFFER, 90, PacketInReason.OFPR_ACTION, 0, 0x0000000000000000, match, data_in)
224          send_message(tcp_socket, msg)
```

Figure 5.3: Packet_In

6 Test Results

In this chapter, the results of the test cases are presented and explained. In Section 5.1 the DELTA test cases were described in detail and in 5.2 the test cases which are executed with the own program were described. These test cases were summarized in Figure 6.1. The test cases are executed in the test environments in the order shown in the table. This table thus serves as a guide and overview for the test executions.

Test Case No.	Attack Description
	DELTA
2.1.010	Malformed Version Number
2.1.020	Corrupted Control Message (Corrupted Content)
2.1.030	Handshake without Hello
2.1.040	Control Message before Hello
2.1.050	Multiple Main Connection Request from Same Switch
	Self-Written Program
#0	Initial Test
#2.1.020_a	Corrupted Control Message (Corrupted Content) during connection setup
#2.1.020_b	Corrupted Control Message (Corrupted Content) during normal operation
#3.1.010	Packet-in Flooding

Figure 6.1: Test cases

As explained in detail in Section 4.2, the test cases were performed in two test environments, the virtual and the table setup. The setup is as follows, the order of the tests were already defined by the Figure 6.1. First, the test cases were performed in the virtual test environment. The goal of the virtual test environment is primarily to check whether the tests are executable in general. After all, if a test is functional there, it is due to the test itself if something does not work during execution on the table setup use case. In addition, this test environment serves as a reference and comparison value for the test execution in practice (on the table setup). After the execution and determination of the results, the tests were performed on the table setup.

The results of the tests performed were described as *FAIL* and *PASS*. This presentation of results was adopted from DELTA because in DELTA the output of a test is *FAIL* or *PASS*. *FAIL* means that the test was vulnerable to the attack, thus the attack was successful. Accordingly, *PASS* indicates when an attack was not successful and thus no vulnerability was identified. The terminology was chosen so that the SDN controller is tested to see if it can withstand attacks, i.e. to discover vulnerabilities and not to carry out attacks with a malicious target.

6.1 Result by Test Case

2.1.010 – Malformed version number

Virtual test environment: The result of the test in which a malformed version number is transmitted is *FAIL*. Figure 6.2 shows how a test or attack is started via DELTA and what the result output looks like.

```
DELTA: A Penetration Testing Framework for Software-Defined Networks

[aA]   - Replaying known attack(s)
[pP]   - Show all known attacks
[cC]   - Show configuration info
[uU]   - Finding an unknown attack
[hH]   - Show Menu
[qQ]   - Quit

"<Tab>" for available commands or [aA] arguments.

Command> A 2.1.010

Start attack!
You can see the detail in WebUI or Log file
ubuntu@10.0.3.152's password:

Test Result: FAIL
If the result is 'FAIL', it is vulnerable to the attack.
Command>
```

Figure 6.2: Attack started via DELTA

The malformed version number is performed after the handshake using OpenFlow packet-in message. This process can be seen in Figure 6.3, which shows the log of the channel agent. Here, the history of the Openflow packets is displayed and it can be seen at which point the attack is initiated and how the SDN controller reacts to it.

Channel Agent Log

```
        Controller IP: 192.168.8.107
        Switch IP: 192.168.0.24
        Cbench Root Path: /home/admin/openflow/oflops/cbench/
[Thread-0] INFO TestControllerCase - [Channel Agent] Start Dummy Switch
[Thread-1] INFO DummySwitch - [Channel Agent] receive HELLO msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive FEATURES_REQUEST msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive GET_CONFIG_REQUEST msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-0] INFO TestControllerCase - [Channel Agent] Handshake completed
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-1] INFO DummySwitch - [Channel Agent] receive STATS_REQ msg
[Thread-0] INFO TestControllerCase - [Channel Agent] 2.1.010 - Malformed Version Number test
[Thread-0] INFO TestControllerCase - * SendPKT | PKT_IN : Hub --> Core = OF1.0
[Thread-0] INFO TestControllerCase - * Test | Send Packet-In msg with OF version 1.0
[Thread-0] INFO TestControllerCase - * Result | No response
[Thread-0] INFO TestControllerCase - * Test | Response: Null
[Thread-0] INFO Interface - [Channel Agent] Closing...
```

Figure 6.3: Channel Agent log

The Agent Manager log, shown in Figure 6.4, displays what the setup looks like as well as the general logging and notification of the result.

Agent Manager Log

```
[Thread-6] INFO AttackConductor - * Test | 2.1.010 - Malformed Version Number (supported but not negotiated version) - Test for controller protection against communication with mismatched OpenFlow versions
[Thread-6] INFO TestControllerCase - Target controller: ONOS 1.13.1
[Thread-6] INFO TestControllerCase - Target controller is starting..
[Thread-1] INFO AttackConductor - Channel agent connected
[Thread-1] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = config,version:1.3,nic:eth0,port:6633,controller_ip:10.0.3.251,switch_ip:10.0.3.79,handler:dummy,cbench:/home/ubuntu/oflops/cbench/
[Thread-6] INFO ONOSHandler - Waiting for ONOS launch..
[Thread-6] INFO TestControllerCase - Listening to switches..
[Thread-6] INFO TestControllerCase - Dummy switch starts
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = startsw
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.010
[Thread-6] INFO TestControllerCase - Send Packet-In msg with OF version 1.0
[Thread-6] INFO TestControllerCase - Response is NULL (expected msg is ERR), FAIL
[Thread-6] INFO AttackConductor - Running Time(s) : 32.346
[Thread-6] INFO AttackConductor - Malformed Version Number (supported but not negotiated version) is done
==========================================================================
```

Figure 6.4: Agent Manager log

According to the Agent Manager log, the result is *FAIL* because the SDN controller response is *NULL* and DELTA specified an OpenFlow Error Message as a prerequisite for the test result *PASS*. A discussion of the result will be performed after the execution at the table setup.

Table Setup In the next step, the test was performed on the table setup, again the result is *FAIL*. Figure 6.5 shows the start and the result of the test in the console.

```
DELTA: A Penetration Testing Framework for Software-Defined Networks

[aA]   - Replaying known attack(s)
[pP]   - Show all known attacks
[cC]   - Show configuration info
[uU]   - Finding an unknown attack
[hH]   - Show Menu
[qQ]   - Quit

"<Tab>" for available commands or [aA] arguments.

Command> a 2.1.010

Start attack!
You can see the detail in WebUI or Log file

Test Result: FAIL
If the result is 'FAIL', it is vulnerable to the attack.
Command>
```

Figure 6.5: Test performed on the table setup

The Agent Manager log is similar to Figure 6.4, except that the SDN controller is the one from the tabletop setup and not one created virtually for testing purposes.

Summary, Evaluation and Measures At this point it becomes clear again that DELTA is not necessarily an all-encompassing solution. Of course, it is a vulnerability if the controller does not react with further actions, such as an error message. An Error message would make it more emphasized that the SDN controller has detected the anomaly and is thus resistant to the test. However, the SDN controller's reaction may well mean that the received packet is not authentic to it or that it violates a security policy and for this reason it is discarded and no action is taken. For this reason, this test and the improvement at this point should only be seen as a recommendation and not as a hard criterion to better protect the SDN controller at this point.

A recommendation is as follows: If the SDN controller receives an OpenFlow message that does not look legitimate or violates a security policy, it is better for the SDN controller to send an error message to the switch instead of simply responding with a NULL response. An error message would signal to the switch that the received packet was rejected and why it was rejected. The switch could then handle the rejected packet accordingly and take further action if necessary. Without an error message, the switch could consider the received packet legitimate and forward it, which could lead to security issues or network problems. An error message would also help the SDN administrator identify the cause of the problem and take appropriate action to protect the network.

2.1.020 – Corrupted Control Message (Corrupted Content)

Virtual Test Environment For an incomprehensible reason, this test could not be performed in the virtual test environment. For this reason, only the test result of the table setup is available.

Table Setup The result of the test in which a corrupted control message (corrupted content) is used is *FAIL*. Figure 6.6 shows how a test or attack is started via DELTA and what the result output looks like.

```
Command> a 2.1.020

Start attack!
You can see the detail in WebUI or Log file

Test Result: FAIL
If the result is 'FAIL', it is vulnerable to the attack.
Command>
```

Figure 6.6: Test with corrupted message control

The Agent Manager log, shown in Figure 6.7, displays what the setup looks like as well as the general logging and notification of the result.

```
[Thread-12] INFO TestControllerCase - Listening to switches..
[Thread-12] INFO TestControllerCase - Dummy switch starts
[Thread-12] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = startsw
[Thread-12] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.020
[Thread-12] INFO TestControllerCase - Send a packet-in message with unknown message type
[Thread-12] INFO TestControllerCase - Response is NULL (expected msg is ERR), FAIL
[Thread-12] INFO AttackConductor - Running Time(s) : 22.999
[Thread-12] INFO AttackConductor - Corrupted Control Message Type is done
==========================================================================
```

Figure 6.7: Agent Manager log

The Agent Manager log shows that the result is *FAIL* because the SDN controller response is *NULL* and DELTA specified an OpenFlow Error Message as a prerequisite for the test result *PASS*.

Summary, Evaluation and Measures The recommendation and evaluation of this result is the same as that already given for test case *2.1.010*. With regard to the non-functioning of the test in the virtual environment and the fact that no logs were available for this, from which one could derive why this test was not feasible,

a deficiency or suggestion for improvement for DELTA has unfortunately also been noticed at this point.

2.1.030 – Handshake without Hello

The test results for the virtual environment and for the table setup are identical and the result is *PASS* in each case. This means that the SDN controller is protected against this attack and has noticed the misbehavior of the switch and therefore no connection was established. Figure 6.8 shows the course of the test on the basis of the Agent Manager log of the table setup.

```
[Thread-9] INFO TestControllerCase - Listening to switches..
[Thread-9] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.030
[Thread-9] INFO TestControllerCase - Dummy switch dosen't send hello message
[Thread-9] INFO TestControllerCase - Check switch connections
[Thread-9] INFO TestControllerCase - Switch not connected, PASS
[Thread-9] INFO AttackConductor - Running Time(s) : 27.465
[Thread-9] INFO AttackConductor - Handshake without Hello Message is done
==========================================================================
```

Figure 6.8: Agent Manager log

2.1.040 – Control Message before Hello

The test results for the virtual environment and for the table setup are identical and the result is *PASS* in each case. This means that the SDN controller is protected against this attack and has noticed the misbehavior of the switch and thus has not performed any further actions with the switch. Figure 6.8 shows an excerpt of the Agent Manager log and the result. Here we can see that the desired response from the SDN controller is to ignore the message from the disconnected switch. In this case, it is appropriate to ignore the message and not send an error message, since no connection has been established and it therefore makes more sense to ignore the message.

```
[Thread-12] INFO TestControllerCase - Listening to switches..
[Thread-12] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.040
[Thread-12] INFO TestControllerCase - Send a packet-in message before handshake
[Thread-12] INFO TestControllerCase - Response is ignored, PASS
[Thread-12] INFO AttackConductor - Running Time(s) : 23.063
[Thread-12] INFO AttackConductor - Control Message before Hello Message (Main Connection) is done
======================================================================
```

Figure 6.9: Agent Manager log

2.1.050 – Multiple Main Connection Request from Same Switch

Virtual Test Environment When attempting to perform multiple main connection requests from the same switch, the result for the virtual test environment is *PASS*. In Figure 6.10, the log extract from the Agent Manager shows that the SDN controller of the virtual test environment rejects the other same switch. Thus, the SDN controller satisfies the desired behavior and does not expose any vulnerabilities.

```
[Thread-6] INFO TestControllerCase - Listening to switches..
[Thread-6] INFO TestControllerCase - Dummy switch starts
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = startsw
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.050
[Thread-6] INFO TestControllerCase - Start another dummy switch
[Thread-6] INFO TestControllerCase - Check switch connections
[Thread-6] INFO TestControllerCase - Reject other same switches, PASS
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = stoptemp
[Thread-6] INFO AttackConductor - Running Time(s) : 27.455
[Thread-6] INFO AttackConductor - Multiple main connection request from same switch is done
==========================================================================
```

Figure 6.10: Agent Manager log

Table Setup The result of running this test on the table setup is *FAIL*. The SDN controller does not reject the other same switch as in the virtual test environment, but accepts it (see Agent Manager log extract in Figure 6.11). This behavior has a vulnerability that should not be underestimated.

```
[Thread-6] INFO TestControllerCase - Listening to switches..
[Thread-6] INFO TestControllerCase - Dummy switch starts
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = startsw
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = 2.1.050
[Thread-6] INFO TestControllerCase - Start another dummy switch
[Thread-6] INFO TestControllerCase - Check switch connections
[Thread-6] INFO TestControllerCase - Accept other same switches, FAIL
[Thread-6] INFO ChannelAgentManager - * SendPKT | CMD : Manager --> Channel agent = stoptemp
[Thread-6] INFO AttackConductor - Running Time(s) : 23.426
[Thread-6] INFO AttackConductor - Multiple main connection request from same switch is done
==========================================================================
```

Figure 6.11: Agent Manager log

Summary, Evaluation and Measures Performing this test revealed a vulnerability in the SDN controller because it accepts multiple connection requests, and resulting connections, from same switch. When a switch sends multiple main connection requests to the same SDN controller, the controller should usually reject all but one. However, if the controller accepts multiple main connections from the same switch

instance, this could cause problems in the network. A vulnerability created by accepting multiple main connections from the same switch instance is the possibility of attacks designed to disrupt or manipulate network communications. For example, by sending multiple main connection requests, an attacker can attempt to perform a Denial-of-Service (DoS) attack or execute a Man-in-The-Middle (MiTM) attack. Accepting multiple main connections can also cause network congestion and affect the operation of the controller, ultimately leading to controller and/or network failure.

Therefore, it is important that the SDN controller ensures that only one main connection is accepted from the same switch instance to ensure the integrity, confidentiality, and availability of the network. One possible solution is for the controller to monitor the source IP address and the source MAC address of the switch and reject the additional requests if there are repeated connection requests from the same switch.

#0 - Initial Test

DELTA provided a GUI as well as a command Line Interface (CLI) to run tests and analyze the results. In order to determine and analyze the results and impact of the test cases performed by the self-written program, the use of Wireshark [27] was necessary. Wireshark has an OpenFlow filter to display the OpenFlow communication between SDN controller and switch.

For the initial test that a connection can be successfully established between the SDN controller and the simulated switch, Wireshark was used to check whether the connection had been established and communication was taking place. Figure 6.12 shows a section of the communication. The switch starts communication with a Hello message and receives one from the SDN controller, after which further communication is established as described in 2.3.

Figure 6.12: Wireshark

The test result for both the virtual test environment and the table setup is *FAIL*, as it was possible to establish a successful connection with the respective SDN controller in both test environments.

Summary, Evaluation and Measures A SDN controller typically captures network topology information by receiving topology information from the SDN switches in the network. If a spoofed switch can successfully connect to the controller, the controller may think that this switch is also part of the network's topology, even though this is not the case. This can lead to incorrect decisions on the network, such as routing or security policy enforcement. In addition, attacks and vulnerability testing can proceed from the mistakenly connected switch. This is evident from the other tests that will be conducted as part of this project. Therefore, it is important that SDN networks are protected against such attacks by implementing mechanisms to authenticate SDN switches and verify the authenticity of topology information.

#2.1.020 – Corrupted Control Message (Corrupted Content)

#2.1.020_a During Connection Setup For both the virtual test environment and the table setup, the result of this test is *PASS*. In both cases, the SDN controller terminates communication.

Summary, Evaluation and Measures It is sufficient to abort the communication without sending an error message, since the corrupted control message was sent directly during the connection setup. This case is similar to the result of *2.1.040*, at this point it is appropriate to ignore the message and not to send an error message, since no connection was established.

#2.1.020_b During Normal Operation The result of the two test environments is also the same in this case. Unfortunately, it cannot be decided exactly whether it is a *PASS* or a *FAIL*. The behavior of the SDN controller is as follows, it does not abort the communication, ignores the switch for a short time and repeats the request in a later cycle.

Summary, Evaluation and Measures Since in this case it is a validated switch in the network, the controller does not abort the communication directly, this is a desired behavior. However, it also does not send an error message to signal that it has noticed the corrupt content and to give reason to fix it, instead it re-sends its request, to which it has received the non-expected response, at a later time. Thus, the decision whether it is a *FAIL* or a *PASS* is not so easy to make. It depends on the requirements of the SDN network and the behavior of the SDN controller. One recommendation would be to send an error message to draw attention to the anomaly. However, this is not a strict requirement to ensure security.

#3.1.010 – Packet-in Flooding

The test result for both the virtual test environment and the table setup is *PASS*. The SDN controller did not respond and could not be disabled. There was no impairments of the SDN network while sending packet-in messages in either test environment.

6.2 Result Overview

An overview of the results can be seen in Figure 6.13.

Test Case No.	Attack Description	Result Virtual Test Environment	Result Table Setup
		DELTA	
2.1.010	Malformed Version Number	FAIL	FAIL
2.1.020	Corrupted Control Message (Corrupted Content)	Test did not work	FAIL
2.1.030	Handshake without Hello	PASS	PASS
2.1.040	Control Message before Hello	PASS	PASS
2.1.050	Multiple Main Connection Request from Same Switch	PASS	FAIL
		Self-Written Program	
#0	Initial Test	FAIL	FAIL
#2.1.020_a	Corrupted Control Message (Corrupted Content) during connection setup	PASS	PASS
#2.1.020_b	Corrupted Control Message (Corrupted Content) during normal operation	Indefinable whether PASS or FAIL (-> depends on use case)	Indefinable whether PASS or FAIL (-> depends on use case)
#3.1.010	Packet-in Flooding	PASS	PASS

Figure 6.13: Test results

Of the test cases in the virtual test environment, out of a total of nine test cases (*#2.1.020_a* and *#2.1.020_b* are each considered a standalone test case), two tests had found vulnerabilities, one failed (*#2.1.020*), one result could not be accurately determined (*#2.1.020_b*), and five tests passed.

In the test cases performed on the table setup, weak points occurred in four places, one result cannot be determined exactly (*#2.1.020_b*) and four tests passed the test.

The test results in the virtual test environment and the table setup are the same with two exceptions. Firstly, test case *2.1.020* did not work in the virtual environment, which in itself does not require any further interpretation. Second, test case *2.1.050* produced different results in the test environments. This result raised more questions, as it is exactly this vulnerability that can be considered critical. There may be several reasons why the two SDN controllers behaved differently. One possibility is that the tabletop SDN controller has other applications installed that are required for its use. Depending on the content of these applications, they may affect the behavior of the SDN controller. Another cause could be that the ONOS versions of the SDN controllers differ and the SDN controller of the virtual environment uses a more recent version than the SDN controller of the tabletop setup. So it is possible that this vulnerability will be fixed with an update of the ONOS version.

7 Conclusion

Several observations were made during the execution of this project. Firstly, regarding the hybrid approach of using both DELTA and a self-written program. Secondly, about the security of the SDN controllers used.

Conclusion of the Hybrid Approach Regarding the quality of DELTA, it can be said that it is a comprehensive framework to easily and efficiently check the SDN controller for vulnerabilities once the setup is complete. However, DELTA also has its limitations, on the one hand it is difficult to understand under which arguments a *FAIL* and *PASS* is decided. This became clear in tests *2.1.010* and *2.1.020*. To improve this point, a multilevel evaluation would perhaps be more appropriate, which classifies the exposure to this vulnerability. Or an additional classification which behavior of the SDN controller is more threatening, because the behavior to ignore the message is more harmless than the behavior to actively act on this message. Another point that has been noticed with DELTA is that it is not very flexible in use. Tests that did not work for reasons that were not clear, were unfortunately not modifiable, to solve the problem. As a result, the handling was often very static and limited.

Analogously, the added value of the self-written program became clear here. The initially intimidating effort to implement the basic functions of a switch paid off in the end in that a dynamically expandable program was available that could perform tests in which DELTA functioned inaccurately or not at all. Dynamic is very important, especially when testing for vulnerabilities. As we have seen from test case *#2.1.020*, there are many variations on how this test could have been run. And you could also modify several messages at this point to see what impact it has on the SDN controller and thus continue the analysis for vulnerabilities in more detail.

In conclusion, DELTA added value and was a good tool for evaluating the SDN controller, but DELTA alone can still be expanded and is not sufficient for a detailed test for vulnerabilities. For this reason, the use of a self-written program that imitates a switch is more recommended, as the application can be made more flexible and versatile in order to perform a reliable vulnerability analysis.

Conclusion of the Vulnerability Assessment By performing the test cases, or attacks, some vulnerabilities were discovered. On one hand, the default ONOS controllers themselves (since they also occurred in the virtual test environment), and on the other hand, the configured SDN controller on the table setup. Recommendations were primarily made for the failed tests, as this vulnerability does not lend itself sufficiently to potentially threatening attacks. However, a non-minor vulnerability was also discovered in the table setup, where it is possible to perform multiple main connections from the same switch (test case *2.1.050*). Solutions were proposed to solve this vulnerability.

7.1 Perspective

Based on this project, a multitude of further projects would be possible. One approach could be to deal more concretely with DELTA and to make intensive efforts to make the test cases modifiable, possibly there is also the possibility to add further test cases to DELTA. A recommended approach would be to extend the self-written program with further test cases or even to create several attack vectors to perform more complex combinations of attacks. Pors [24] described that it is a promising possibility to attack the SDN controller using error messages. This approach has already been prepared by implementing in the program all the error messages that can be sent from a switch to a SDN controller, so that they correspond to a real switch. Accordingly, it would be possible to embed more complex attacks in the error messages directly at this point. Another approach, based on the results of this project, would be to increase the security of the SDN controller by implementing the measures proposed here. In general, however, the hybrid approach pursued here is not recommended for further

projects, since the handling of both approaches is too different to be able to expand both efficiently at the same time.

Bibliography

[1] Abdelhadi Azzouni et al. ‘Fingerprinting OpenFlow controllers: The first step to attack an SDN control plane’. In: (Nov. 2016).

[2] Ryu SDN Framework Community. *Ryu SDN framework*. https://ryu-sdn.org/.

[3] DELTA. *DELTA: SDN SECURITY EVALUATION FRAMEWORK*. https://github.com/seungsoo-lee/DELTA. May 2021.

[4] Eder Leao Fernandes. ‘Software Switch 1.3: An experimenter-friendy OpenFlow implementation’. PhD thesis. Universidade Estadual de Campinas, March, 2015.

[5] Project Floodlight. https://groups.io/g/floodlight. Feb. 2016.

[6] Linux Foundation. *OpenvSwitch*. https://www.openvswitch.org/.

[7] The Open Networking Foundation. *OpenFlow Switch Specification*. Mar. 2015.

[8] Nicholas Gray et al. ‘Simulation Framework for Distributed SDN-Controller Architectures in OMNeT++’. In: vol. 191. Jan. 2017, pp. 3–18. DOI: 10.1007/978-3-319-52712-3_1.

[9] CORE Grou. *Tischaufbau des SecVI Demonstrators*. https://trac.core-rg.de/trac/wiki/SecVI$_{T}ischau$

[10] Huawei. *SDN Networking Type of Messages and Channel Connection In OpenFlow*. https://forum.huawei.com/enterprise/en/sdn-networking-type-of-messages-and-channel-connection-in-openflow/thread/796175-871. Apr. 2022.

[11] Dominik Klein and Michael Jarschel. ‘An OpenFlow extension for the OMNeT++ INET framework’. In: Jan. 2013, pp. 322–329. DOI: 10.4108/simutools.2013.251722.

[12] Rowan Klöti, Vasileios Kotronis and Paul Smith. 'OpenFlow: A security analysis'. In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. 2013, pp. 1–6. DOI: 10.1109/ICNP.2013.6733671.

[13] Kytos. *Python OpenFlow*. https://docs.kytos.io/kytos/developer/pyof/. Jan. 2021.

[14] Seungsoo Lee et al. 'A comprehensive security assessment framework for software-defined networks'. In: *Computers & Security* 91 (2020), p. 101720.

[15] Seungsoo Lee et al. 'Delta: A security assessment framework for software-defined networks.' In: *NDSS*. 2017.

[16] Hewlett Packard Enterprise Development LP. *OpenFlow channel*. https://techhub.hpe.com/eginfol
4024$_{o}penflow_{c}g/content/499752688.htm$. 2017.

[17] Jahanzaib Malik et al. 'Hybrid Deep Learning: An Efficient Reconnaissance and Surveillance Detection Mechanism in SDN'. In: *IEEE Access* 8 (2020), pp. 134695–134706. DOI: 10.1109/ACCESS.2020.3009849.

[18] Josias Montag. *Software Defined Networking mit OpenFlow*. 2013.

[19] Moris. *SDN-Switches: Erläuterung und Vorteile*. June 2022.

[20] ONF. *DELTA*. https://opennetworking.org/incubator-projects/delta/. May 2021.

[21] ONOS. *ONOS Releases*. https://wiki.onosproject.org/display/ONOS/Downloads. Mar. 2021.

[22] OpenDaylight. *OpenDaylight*. https://www.opendaylight.org/.

[23] Gregory Pickett. *Abusing Software Defined Networks*. 2014.

[24] Marlou Pors. 'Investigating current state Security of OpenFlow Networks: Focusing on the control-data plane communications'. In: 2017.

[25] Open Network Operating System. *Open Network Operating System (ONOS)*. https://opennetworking.org/onos/.

[26] Benjamin E. Ujcich. 'Securing the software-defined networking control plane by using control and data dependency techniques'. PhD thesis. University of Illinois at Urbana-Champaign, 2020-07-16.

[27] Wireshark. *Wireshark*. https://www.wireshark.org/.

[28] Andi Xu et al. 'Improving Efficiency of Authenticated OpenFlow Handshake Using Coprocessors'. In: *2016 8th International Conference on Information Technology in Medicine and Education (ITME)* (2016), pp. 576–580.

[29] Nian Xue, Xin Huang and Jie Zhang. 'S2Net: A Security Framework for Software Defined Intelligent Building Networks'. In: Aug. 2016, pp. 654–661. DOI: 10.1109/TrustCom.2016.0122.