INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Monitoring and Mitigating attacks in the environment of Software Defined Networks

Christoph Girstenbrei
Alex Marczinek

INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

Monitoring and Mitigating attacks in the environment of Software Defined Networks

Christoph Girstenbrei

Alex Marczinek

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Dr. Nils gentschen Felde Tobias Guggemos Daniel Migault (Ericsson)
Abgabetermin:	12. Dezember 2016

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 12. Dezember 2016

. . . .

(Unterschrift des Kandidaten)

Preface

This thesis is the result of a collaborative work. As such, the source code authored during this work is the result of joint effort.
In this thesis, chapters written by the other contributor are clearly marked as direct citations. As chapters 1,3 and 7 have been written by both authors, the author of each paragraph is marked on the margin.

Abstract

In recent times, Software Defined Network (SDN) have quickly gained popularity. The opened capabilities and addressing of shortcomings in traditional networks has sparked a lot of interest. But considering the extent of testing regular networks are over, SDNs have to make up a lot of ground. Especially due to new attacks on network clients and networks itself, a reliable and adaptive strategy is necessary. Having insight into the network and access to reliable information is one of the most important properties of a modern network. The intent of this thesis is to address the first step of monitoring data, making it available to other components and alert on possible attacks to enable a reactive and flexible network in context of an SDN environment. Different techniques of gathering, analyzing and output data will be proposed and shown with an example implementation. Comparison between these techniques will present their advantages and disadvantages, providing information on which tool to choose for which type of data or attack.

Abstract

Software Defined Networks are a technology that is quickly becoming popular. It enables new network functionality through its standardized interface and the decoupling of the data and control plane.

Yet, also attacks on networks are as popular as ever, with attacks disrupting the operation of even large and well prepared companies. These attacks often are Denial of Service attacks. The architecture of Software Defined Networks also has exploitable vulnerabilities

To help overcome these issues, a detection and mitigation mechanism based on SDN within a reproducible test environment is proposed. This builds a foundation for research on monitoring and mitigation in software defined environments.

This test setup is then used for the implementation and evaluation of TCP SYN Flood, Distributed TCP SYN Flood and Flow Flooding attacks. The results show that by monitoring the network through an Intrusion Detection System and using statistics of the flows, the attacks can be detected and this information can be used to create flows that are capable of mitigating the attacks.

Contents

1. Introduction

Traditionally, networking technology is a field dominated by the vendors.[JMD14] describes that each vendor supplies its own proprietary hardware and closed soft- and firmware, offering little in the way of flexibility, innovation and adaptation to the steadily changing system and service requirements of enterprises and customers alike. As such, the evolution of networking technology has been slow, and we have yet to see a major contender arise to topple the status quo.

One contender that might have the potential to change the current state of networking, is Software Defined Networking. The defining point of SDN is its decoupled nature, separating control and packet forwarding functionalities into separate layers and centralizing network intelligence and state, thereby allowing for increased functionality, automation, control and programmability [MBR16; ONF14].

OpenFlow is a SDN protocol that has lead to the technology receiving strong interest from various large companies, such as Facebook, Google and Microsoft[GB14], as well as sparking interest from research communities [JMD14]. OpenFlow provides standardized interfaces to communicate with the forwarding devices [McK+08], thus being the basis for an architecture with a centralized controller.

With this new architecture, the forwarding and routing decisions of the devices can be programmed by Network Applications through the controller. They run fully in software and allow every forwarding device to effectively resume responsibilities traditionally handled by specialized middleboxes, resulting in a very flexible network that can better meet the demands of current networks.[GB14]

Security in our digital world is a topic everybody has already heard of. Reports of people managing to compromise or take out the networks of large companies make it to major newspapers regularly. Examples are the attack on Sony Pictures Entertainment in 2014 [BBC14] or, more recently, the DDoS attack on Dyn, a company proving DNS services, which brought down large parts of America's Internet including sites such as Twitter, Netflix and Reddit, in october this year[Gua16].

Therefore the advent of a rather new networking technology has served as motivation for exploring the field of security in Software Defined Networks. By using Flows to influence the way traffic is forwarded through the network and by taking advantage of the advanced statistics OpenFlow devices can collect, new techniques for detecting threats to the network are investigated. At the same time Flows provide a vast amount of possibilities to block malicious packets at the forwarding devices.

To this end, a virtual, OpenFlow based testing environment is created, presented and subjected to a number of attacks. With a Virtual Machine (VM) as an automated testing environment, it is possible to set up the same environment repeatedly and reliably. Modeling a real network as close as possible, Linux Containers (LXCs) are used to simulate network clients. This allows, in comparison to other approaches like Mininet, for a fully functional system stack including system services and maximum separation between the simulated hosts, including the file system. After setting up an environment, attacks can be conducted and observed, data gathered and the effects measured. After that, a defense strategy can be established and tested to prove its effectiveness.

To investigate capabilities in an SDN, this thesis takes a look at two different kinds of attacks. The first kind is attacks commonly found in regular network environments. As the new network technology has to withstand already existing attacks, it can prove its resistance to those. The second kind is new attacks that specifically target the infrastructure of the SDN environment. As the network itself has vulnerabilities, SDNs will have to face and defend against such attacks.

An attack commonly found in all networks is DDoS (Distributed Denial of Service). The aim is to try to prevent legitimate users from accessing a service, by consuming all the available bandwidth or resources of a system and therefore rendering it incapable of serving the requests of legitimate clients. There is a plethora of tools like Low Orbit Ion Cannon or Stacheldraht readily available online that allow anyone with hardly any prior knowledge, to execute a DDoS attack.[Lau+00; ZJT13] Given its ease of execution and effectiveness, it comes as little surprise that DDoS currently is one of the most common attacks on networks. [Sec16].

An example for the second category is the so called flow flooding attack. Aiming to modify and flood the flow tables administered by an SDN controller, this attack carries the possibility of a widespread impact, resulting in an unstable and unusable network. As it is a new attack, the tools are not as available compared to Distributed Denial of Service (DDoS) tools. But as SDN grows in prominence, tools will also be more readily available and mature. SDN will have to be prepared for these challenges.

The goal of this thesis is to develop an SDN aware application stack, capable of detecting and mitigating a variety of attacks. In order to achieve this, intermediate steps include the development and implementation of a test setup to simulate a network. Next, an application stack is proposed to handle monitoring and mitigation of the same network. Then, different attacks are executed against the setup to test and prove its capabilities and show its weak points.

The thesis is organized as follows. In chapter 2 Software Defined Networks and Attacks in the network are explained in more detail. In chapter 3 related background literature in the field is discussed. In chapter 4 the implementation of a virtual test environment is discussed in detail and the tools used in later chapters are introduced. Chapter 5 explores techniques to monitor traffic in SDN and the detection of attacks. In chapter 6 strategies to defend against attacks are introduced and mitigation strategies for a selection of Denial of Service attacks are implemented and evaluated in the environment of the test setup. Finally, chapter 7 draws conclusions.

2. Background

This chapter provides a more detailed overview of the core concepts used in this thesis.

More specifically section 2.1 discusses Software Defined Networking and explains why there is a demand for this technology, how it differs from the networks predominantly in use today and how those differences are implemented and can improve the status quo.

In section 2.2 attacks in the environment of Software Defined Networks are explored. Hereby attacks that can also be found in traditional networks are introduced, as well as attacks that target vulnerabilities of the SDN environment specifically.

2.1. Software Defined Networking

Born from an academic research project, SDN has already had a huge impact on the networking industry. Its real success has started with the release of the OpenFlow protocol. With large companies such as Deutsche Telekom, Facebook, Google, Microsoft, Verizon, and Yahoo! being responsible for the OpenFlow standard, the technology is advancing rapidly and while other protocols exist and are being developed, such as ForCES, OpenFlow still remains the most widely used.[GB14]

The differences, in the infrastructure between an SDN device and a traditional Internet switch, already show an integral part of the SDN design. To this end, the following gives a quick overview of the architecture of a traditional switch.

2.1.1. Traditional Infrastructure

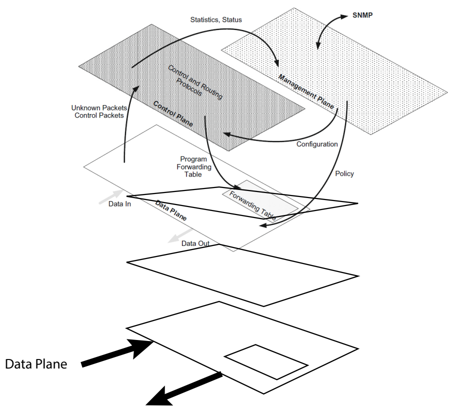

Figure 2.1.: Roles of the control, management, and data planes. From [GB14]

The switching functions can be divided into three categories. As these categories are capable of horizontal communication with other entities within the same category as well as vertical communication with other categories, they are usually represented as three planes as shown in Figure 2.1: the data plane, the control plane and the management plane.

Data Plane

As Goransson et al. mentions, the data plane is comprised of its various ports plus a forwarding table and is in control of forwarding the received packets, as well as packet buffering, scheduling and header modifications. The forwarding table contains the rules dictating how a packet has to be forwarded. Packets, whose header information can be found in the forwarding table, can be forwarded, and its header, if necessary, modified without the help from the other planes. However, in the case of yet unknown packets and in the case of protocol packets, the involvement of the control plane becomes necessary.[GB14]

Control Plane

The main function of the control plane is to keep the forwarding table, which resides in the data plane, up to date. In order to implement new network policies, the exchange of management information between devices is necessary. Thus the control plane implements many control and routing protocols, such as Spanning Tree Protocol (STP), Shortest-Path Bridging (SPB) and Routing Information Protocol (RIP), necessary for managing the forwarding table and thus also the topology of the network. One of the most basic but without a doubt most important is MAC-Learning. On the arrival of a packet the switch learns, based on the source Media-Access-Control (MAC) address, on which port the device exists and creates a rule in the forwarding table. Based on this rule, future packets sent to that MAC address, can be forwarded to the appropriate port. [GB14]

Management Plane

To change the behavior of the aforementioned protocols, the management plane can interact with the control plane, as well as trigger changes in the data plane, such as changing or adding entries in the forwarding table. Further it can query statistic and status reports from the control plane. To make use of these functions, a network administrator uses some sort of network management protocol. A well known example for such a protocol is the Simple Network Management Protocol (SNMP).[GB14]

2.1.2. Why SDN?

In comparison to this architecture, where the control plane and data plane is tightly coupled together inside the device, which was a design choice due to the urge of having very resilient networks, SDN takes a different approach. SDN breaks this vertical integration by taking the control plane off the device.

The reasons behind this decision lie in the issues that the old highly decentralized architecture causes. As a study conducted by Benson et al. shows, the networks have become increasingly complex to manage [BAM09]. Given this complexity, it is only a natural consequence that misconfigured devices are a common occurrence.

As a single misconfigured device can already have a severe impact on network performance, network administrators have tried to facilitate the management by using specialized hardware, so called middleboxes. A middlebox performs a single task, e.g. a firewall or deep packet inspection.[Kre+15] This takes away the burden of having to implement the policy by configuring the network device directly. However, as discussed in [She+12], they have to be configured through vendor specific commands which, combined with the large amount of middleboxes used, only shifts the problem instead of solving it.

2.1.3. SDN Infrastructure

As mentioned previously, the control plane has been separated from the data plane.

This results in an architecture with three separate layers, as depicted in Figure 2.2: The forwarding devices, the SDN controller and the network applications.

In the following, the architecture is presented in more detail in a bottom-up approach.

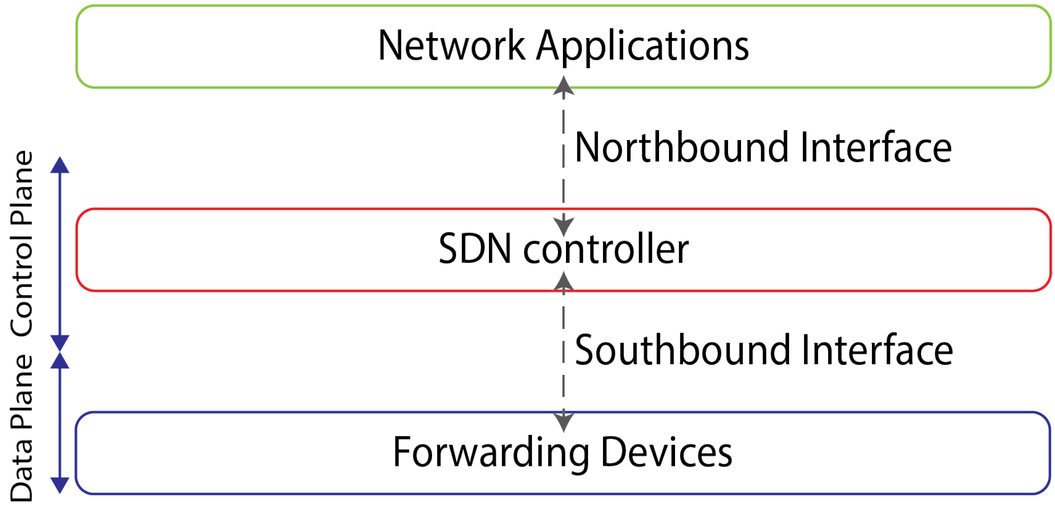

Figure 2.2.: SDN architecture adapted from [Kre+15]

Forwarding Devices

The devices are simple forwarding devices that only know how to handle incoming packets by extracting the characteristics, such as MAC address or IP address, from the header of an incoming packet. This information gets matched against the forwarding table and, if a matching entry is found, the associated action or actions, such as to forward the packet to a specific port, sending it to the controller for further inspection, duplicating it and sending it out on various ports or to simply drop the packet, are executed.[GB14]

Interfaces

The Southbound Interface presents a uniform API for communication between the forwarding devices and the controller. OpenFlow is currently the most well-known Southbound API and standardizes the way the controller should interact with the devices.

The Northbound Interface allows the network applications to talk to the controller. Details of the API are strongly dependent on the controller used. Commonly a REST or an ad-hoc

API is used.

SDN controller

The controller resides on a remote machine in the network. While logically always a single entity, he can actually be distributed over multiple machines, allowing for redundancy and adding stability.

To provide another layer of abstraction, the controller uses a Network Operating System (NOS) facilitating access to the underlying low-level devices and resources, comparable to the abstraction an Operating System provides. This is a clear step forward compared to traditional network management that still relies on low-level interaction with the network device for configuration and management.

There are various Network Operating Systems available with different properties, using different programming languages, such as OpendayLight, which is programmed in Java and uses a distributed controller, or POX, which uses Python as the high-level language and uses a centralized architecture. However as POX currently does not support the newer OpenFlow versions, Ryu provides by far more possibilities to configure the forwarding devices and allows access to most features.

The roles of the controller are the discovery of devices joining or leaving the network and providing network topology information, as well as managing the flows in the forwarding tables. These high-level languages, combined with the provided services, make it much easier for the SDN developer to manage the network and implement new network policies.[Kre+15]

Network Applications

To add additional functionality to the network, services such as MAC-Learning, Routing or an Intrusion Detection System (IDS) can be added. They are fully implemented in software and can run on any machine in the network and communicate with the controller via the Northbound API, through which it can receive alerts, such as a device failure or information about the load, from the controller. Yet can also receive alerts from external sources. Usually the network applications reacts based on these alerts and sends commands to the controller to change the flows of the devices. Thus e.g. an application can receive an alert about suspicious activity in the network and decide to redirect the traffic to an application specialized in detailed inspection. [GB14]

2.1.4. Flows

The data structures in an SDN device are the flow tables and they replace the forwarding table in a traditional device. While at least one flow table is mandatory, there are usually multiple flow tables available. Flow tables resemble traditional forwarding tables in functionality, but are more generic than their counterpart.

The individual entries in a flow table are called flows. They each have a priority associated to them and consist of match fields and actions, as seen in Figure 2.3.

These attributes describe the way a packet or set of packets takes from one endpoint to another. Packets arriving at the device are matched against the entries of the first flow table from the highest to the lowest priority. The first entry, whose match field criteria are fully satisfied is chosen and the actions of that entry are executed.

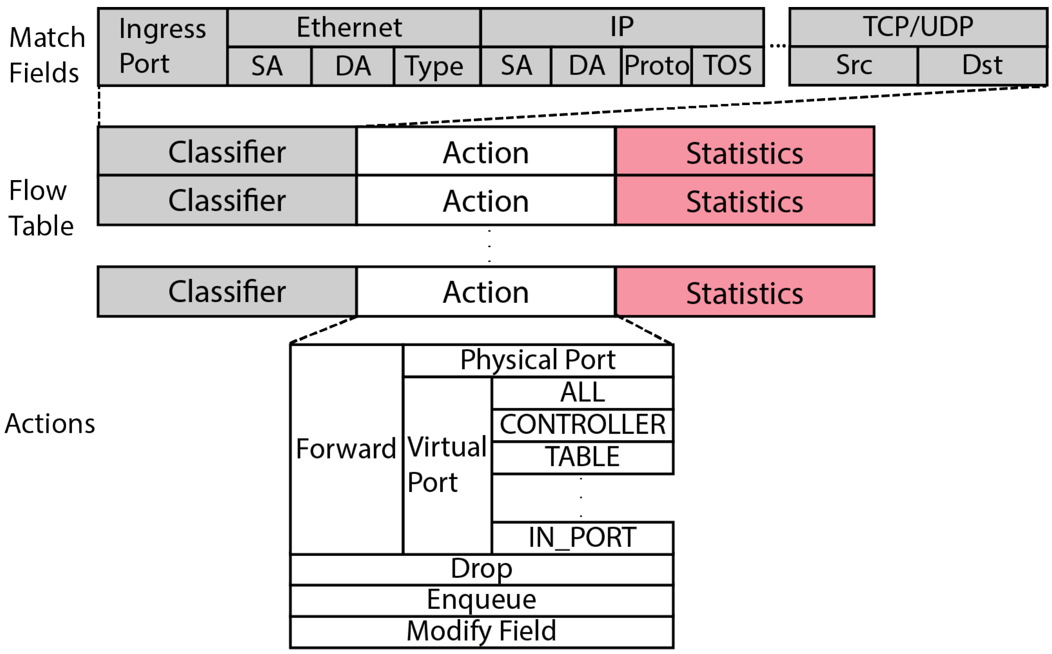

Figure 2.3.: Flow Table and its match fields and actions. Based on [NG13]

By default all possible match criteria are wild-carded, this means any value fulfills it. The fields of interest can then be set to only match a specific value or can be partially wild-carded to match a range of values.

Though the available match fields, e.g. source MAC address and destination IP address, are depending on the implemented protocol, today most devices offer an extensive amount of match fields, giving the SDN developer a lot of flexibility and possibilities in forwarding the traffic. [GB14]

Should no entry match, it is a miss and the packet is discarded. However it is common practice to include a fully wild-carded entry at the lowest priority, sending a miss to the controller instead.

Other possible actions of flow entries, amongst others, are forwarding the packet to an outgoing port, to drop the packet, to send it to another flow table or to a special table like the group table that provides additional functionality, such as duplicating a packet and forwarding it to multiple outgoing ports.

Additionally a flow saves statistics about its usage, such as the amount of packets that has matched to this entry, that can later be queried by the controller.[Kre+15]

2.2. Attacks

Software Defined Networks can, just as traditional networks, be the target of attacks. Denial of Service and Man in the Middle attacks are some of the most prominent examples. But while these attacks apply to SDNs just as much as to traditional networks, the new architecture also opens up new attack vectors.

One thing that stands out in SDNs is the controller. With all the management decisions and the installation of the flow rules relying on the controller, it seems like a prominent target. It helps that the design of SDN allows for a logically centralized but physically distributed controller. However, given its remote nature, it still proves to be a potential bottleneck in the network that can be exploited.

Another security concern is the integrity of the communication on the Southbound and Northbound Interface. Attackers that manage to tamper with the control packets between the controller and the switches or even create fake packets from scratch, can gain full control over the network. Equally important is the communication between the network applications and the controller on the Northbound. While this interface, depending on the implementation of the controller, is potentially not quite as powerful, as the Southbound one, an attack can have the very same consequences.

But not only the management components themselves can be the target of an attacker. The devices, in particular the flow tables, are of interest too. By tampering packets, an attacker can attempt to trick the controller into creating malicious flows that destabilize the whole network. [AX15; LMK16]

In subsection 2.2.2 attacks using these attack vectors are introduced and in subsection 2.2.3 Denial of Service attacks are treated.

2.2.1. IP Spoofing

A security concern, due to the uncontrolled nature of the Internet, is IP Spoofing. As the source of a packet is not verified[YBX11], it is very easy to forge the IP Address and pretend to be someone else, just by changing the source IP Address field in the header.[Tan03]

This enables attacks, like the notorious DNS amplification attack, where the attacker pretends to be the actual target of the attack, causing the DNS server to send large replies to the victim instead of the actual source of the query.[VE06]

On the other hand it conceals the origin of an attack. Being unable to associate malicious packets to an IP Address makes it very hard to trace and stop an attack[Tan03].

As such IP Spoofing is a technique, that many attacks rely on, either as a basic requirement for the attack or as a tool to improve the effectiveness of the attack.

2.2.2. SDN specific attacks

The following shows how the attacks vectors, mentioned previously, can be exploited. To this end an attack, which targets the flow table of a switch, attacks targeting the services provided by the controller, and an attack targeting the communication channel is introduced.

Flow Table Flooding

Technically speaking this attack can be classified as a Denial of Service attack. But what makes this attack unique, is that it targets the flow tables used by the SDN forwarding devices.

By trying to find a pattern in the way the controller installs new flow entries, the attack tries to launch a stream of packets that generates as many new flow entries as possible. As the amount of flow entries a flow table can store is limited, this will ultimately lead to the flow table overflowing and flow entries that are vital to the forwarding of the traffic, can no longer be installed.

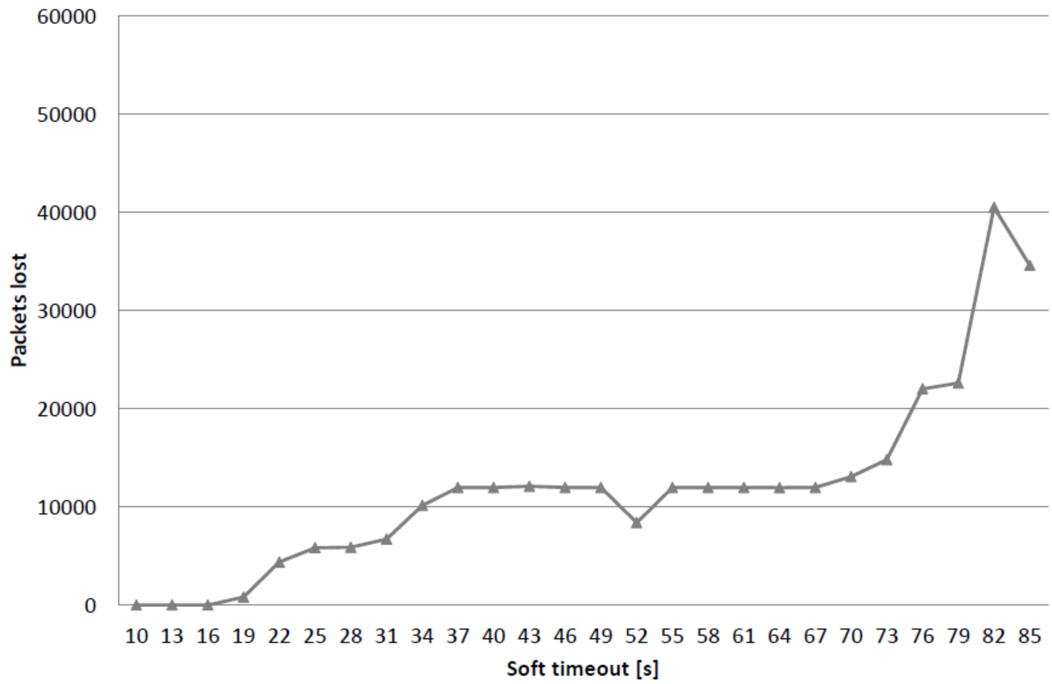

Figure 2.4.: Packet drop due to flow flooding at 100 Mbps. From [KKS13]

Figure 2.4 shows the amount of packet dropped during a flow table flood in regard to the soft timeout for flows, which determines the amount of time a flow may be inactive before it is removed automatically. [KKS13]

Controller

[Hon+15] shows an attack that can seriously compromise one of the core features of the SDN controller. By using a kind of spoofing attack, it misleads the Topology Management Service of the controller. As the controller automatically checks for hosts that have migrated to a new location without further verification, packets can be spoofed to trick the controller into thinking that the target has changed location. Thus future traffic is sent to the attacker's location where, for example in the case of a web server, the website could be impersonated.

Another service that is vulnerable is the Link Discovery Service. As the authentication of the Link Layer Discovery Protocol (LLDP) packets can easily be decoded or reconstructed from the source code of the controller, fake LLDP packets can be crafted. Through these packets actually non-existing links can be created between two switches. As the discovery of a new link also leads to an update of the Shortest Route, this can allow the attacker to introduce a compromised host into the fake link and perform a Man-In-The-Middle attack.[Hon+15]

Communication Channel

As discussed in [Shi+13], the communication channel between the switches and the controller is vulnerable to a saturation attack. As every packet without a matching flow entry is sent to the controller, this channel can quickly become a bottleneck. A DDoS attack designed to cause flow table misses can cause the controller to fall behind in handling these misses and cause the buffer to overflow, resulting in dropped packets. The result is very similar to what happens in a Flow Table Flood, once the flow table is full.

By placing a device between the controller and the switches, for example through a LLDP vulnerability, as discussed earlier, the attacker can gain full control of the network. If Transport Layer Security (TLS) is not implemented the attacker can freely modify or insert flow rules. As the messages from the switches to the control can be intercepted as well the controller will not even notice that something is amiss. Even if TLS is used potentially large parts of the network can be taken down by dropping the packets from the switch and controller. [BCS13]

2.2.3. Denial of Service

A Denial Of Service (DoS) attack is characterized by the explicit attempt to, as the name implies, deny access to a targeted system and prevent legitimate users from using the desired services.

The most common type of DoS attack performed is to flood the target with superfluous requests until its available bandwidth or resources are fully consumed and it can no longer handle regular requests, therefore making the service inaccessible.

Another less common method is a vulnerability attack. This attack abuses a vulnerability in the targeted system and sends malformed packets that lead to the services being unable to perform their intended purposes.[Lau+00]

Distributed Denial of Service

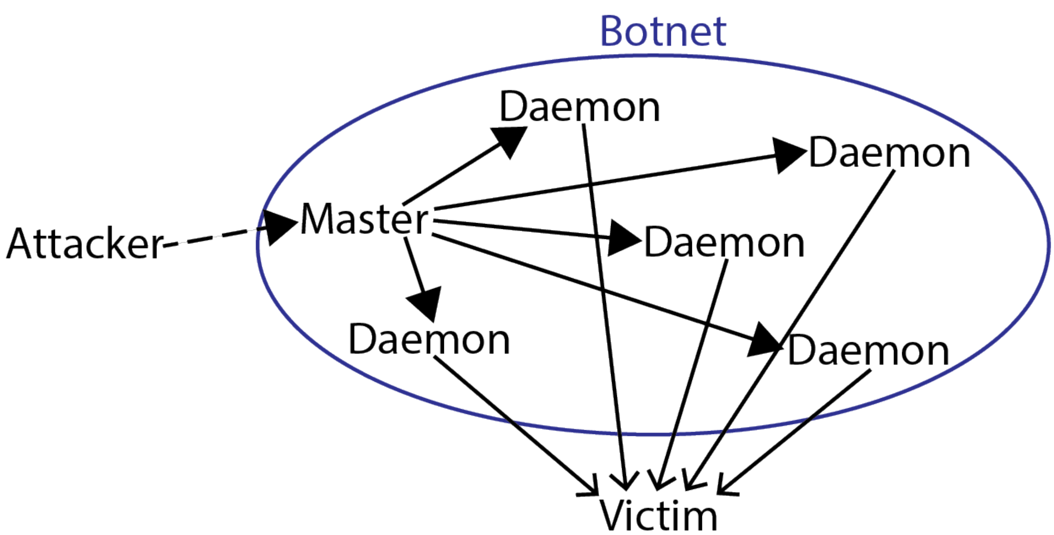

Figure 2.5.: Components of a DDoS attack. Based on [Lau+00]

Most commonly a DoS attack is executed in a distributed fashion and is then called DDoS.

The distributed version is usually executed by a botnet, comprised of computers infected with malware and orchestrated remotely, under the command of the attacker. As seen in Figure 2.5, the attacker coordinates the attack, by sending the attack-command to the master system in the botnet. As a result, the master, who has an overview of the complete botnet, spreads the command to all the other infected systems, usually called Daemon, who then execute the actual attack.

The main benefit of using the master and daemon setup is the concealment of the real culprit behind the attack. While it appears simple to execute the attack, the real work consists of infiltrating enough systems to build a large botnet and searching for vulnerabilities or bottlenecks in the victim's network.

The size of the botnet is often the deciding factor for the success of the attack, as the bandwidth the attack can generate scales linearly with the size of the botnet. This of course directly correlates with the amount of resources the attack can consume.[Lau+00]

The main reason why DDoS is even possible, is that the Internet is not designed to control traffic, but moving packets from source to destination efficiently. This means the intermediate network only does the minimum effort necessary for transporting the packets. However, this design allows for DDoS attacks being possible if only one member in the communication misbehaves, as malicious packets can reach the target without being checked for integrity.[MR04]

TCP Syn Flood

The TCP protocol relies on a three-way handshake to establish a connection, as shown in Figure 2.6. The following shows how this three-way handshake can be exploited to make a system unreachable.

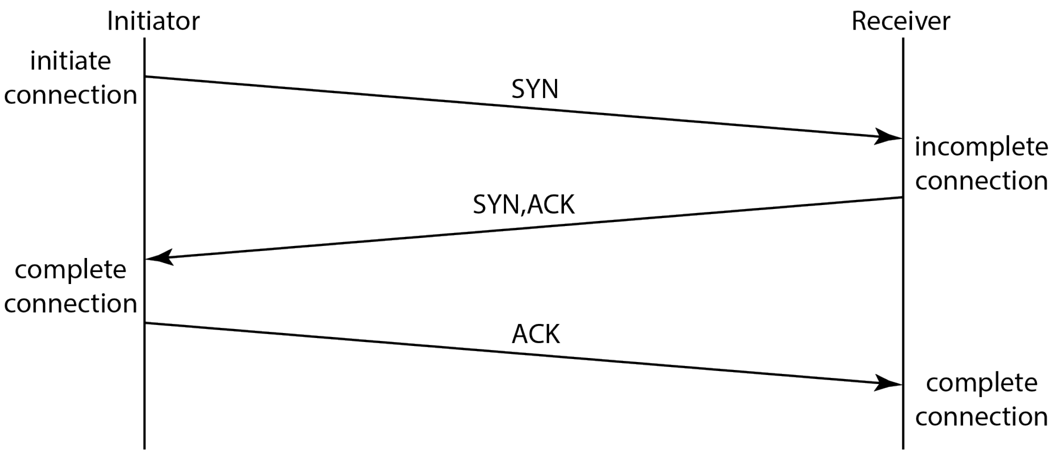

Figure 2.6.: TCP three-way handshake.

On receiving the TCP SYN packet, the receiver replies with a SYN,ACK and waits for the corresponding ACK to complete the establishment of the connection. However, in the mean time he has to allocate resources to save the state of the communication with the initiator.

An attacker can abuse this by never sending the final ACK and thus creating a half-open connection that binds resources of the victim. In a TCP SYN Flood the attacker sends as many SYN requests as possible in order to exhaust all the available resources, rendering the victim unable to answer legitimate connection attempts.[Lem+02]

As shown in [Lem+02], a machine that can have up to 1024 incomplete connections per socket and uses the standard TCP time out of 511 seconds until it drops unsuccessful connections, can be exhausted very easily.

As the limit of 1024 incomplete connections is reached, the machine drops an old incomplete connection to make room for the new connection attempt. If the average time for a packet to make a round trip is 100ms, a Syn Flood attack with a bandwidth of just 4MB/second is enough to completely clear and refill the saved open connections in the mean time. Thus once the ACK arrives there is no record of the corresponding SYN and the connection cannot be established.

Distributed TCP Syn Flood

While the attack remains exactly the same as the non distributed TCP Syn Flood, this attack is far more challenging to defend against.

First of all the use of a botnet makes attacks with an enormous amount of traffic possible. For example Verisign, a company that offers DDoS protections, reports an attack with a peak of approximately 60 Gigabits per second at 150 Million packets per second in Q3 2016 [Ver16]. This makes even systems that have a large pool of resources available susceptible to this attack.

But even more important, the attack packets, especially if combined with IP Spoofing, can no longer be grouped by source IP Address, where a large amount of SYN requests from a single source is a very suspicious behavior. This requires a different defense strategy to mitigate the attack.

3. Related Work

Security in Software Defined Networks is a topic that has garnered a lot of interest recently. The following presents an overview of the literature in the field, related to the work in this thesis.

The setup used in this thesis is heavily based on LXCs to simulate and separate network components within the VM. In [GDK], the method of encapsulated VMs is used. This allows for a more complete component separation, especially by removing the dependency on a shared kernel across LXCs. As the separation capabilities of LXC technology are sufficient to simulate multiple network clients and the kernel dependency does not pose a problem in this setup, the advantages of LXC technology, namely a lightweight isolation approach and therefore fast startup and reaction times, outweigh the advantage of paravirtualization or full virtualization.

Introduced by the monitoring concept of OpenFlow itself, new problems arise. One of them is due to the pull based nature of the protocol, since a gathering component has to actively collect information from the network. This asks for a balance between detailed monitoring and network overhead, a possible solution for this problem is proposed in [Cho+14]. Implementing an abstraction layer between the SDN controller and network monitoring enables for a more intelligent and adaptive monitoring of components than the one currently implemented in the setup of this thesis.

Seen from this angle, scalability and distributability of the monitoring logic in large scale network is necessary. Evaluating the OpenFlow (OF) native approach in comparison with a strategy based on sFlow is done in [Gio+14]. Demonstrating the problem by overloading the SDN control plane with monitoring traffic and comparing the results of both methods with real and high volume traffic, the paper uses an entropy based method to analyze data. As the network used in this setup is considered small, the issue of overloading the system with monitoring traffic is not present, especially due to the low network base load. To implement a similar network on a larger scale successfully, these problems have to be taken into consideration.

A major problem for the security of networks is IP Spoofing. Yao et al. have proposed a mechanism based on the OpenFlow architecture that tries to detect spoofed IP Addresses called VAVE (Virtual source Address Validation Edge). This is an improvement of the system proposed by IETF, SAVI (Source Address Validation Improvements) [BG13]. By forming a perimeter of OpenFlow devices around the network, every packet that comes from a legacy device or from outside of the network is checked. Packets that arrive from outside of the network with an IP Address that is in the IP Address range of the network can then be safely declared as spoofed and dropped. While this technique is useful to prevent attackers from spoofing IP Addresses of systems inside of the network, it cannot prevent any attacks that rely on mostly spoofing random IP Addresses. [YBX11]

AVANT-GUARD[Shi+13] uses SDN switches as SYN proxies. When receiving SYN requests from a new source, the switch starts the TCP handshake using SYN Cookies. Only if the client successfully completes the handshake, the switch installs a flow to forward subsequents packets from that source and completes the handshake with the original destination[Amb+15]. But [Amb+15] also analyzes that AVANT-GUARD is susceptible to a buffer saturation attack, as the switch has to save state to migrate the TCP handshake to the destination.

Cui et al. [Cui+16] propose a system based on 4 modules: Attack detection trigger, detection, traceback and mitigation. They propose a new method to trigger the detection of an attack. Instead of periodically checking, they monitor the amount of packets sent to the controller to start the attack detection, which only then starts gathering the flow stats and searches for malicious entries which trigger mitigation. For their test environment they use mininet. While mininet is even more lightweight than the virtualization via LXC, mininet uses a shared file system and PID space. To guarantee a more flexible host environment, LXC was chosen for this thesis.

4. Test Setup

This chapter aims to provide on one hand a high level overview enabling an understanding of conceptual parts, component interactions and information flow. It shows the basis and concepts on which the setup is implemented on. After providing a design, it will then on the other hand show implementation details giving insight into important parts of components. The parts and software used will be explained and placed in context of the design concept. At the end, advantages and disadvantages of the setup will be compared.

4.1. Concept

To develop a concept for the test setup, 3 basic goals have to be met. The first one is detecting and mitigating attacks in an SDN environment. The second goal is grouping every functionality into encapsulated and movable components following the Single Responsibility Principle (SRP) [Mar03, pp. 95-98]. This allows a more complex environment and separates concerns. Third, all communication paths shall be defined clearly, use Ethernet/Internet Protocol (IP) and be shaped appropriately for the type of information transmitted. This is especially important in a network environment, as the components are not necessarily inside the same virtual or physical machine. Communication can happen across different base media, but the standardization in place by using the IP protocol ensures available communication everywhere and, if necessary, in between every component. Figure 4.1 shows the underlying idea in the application design.

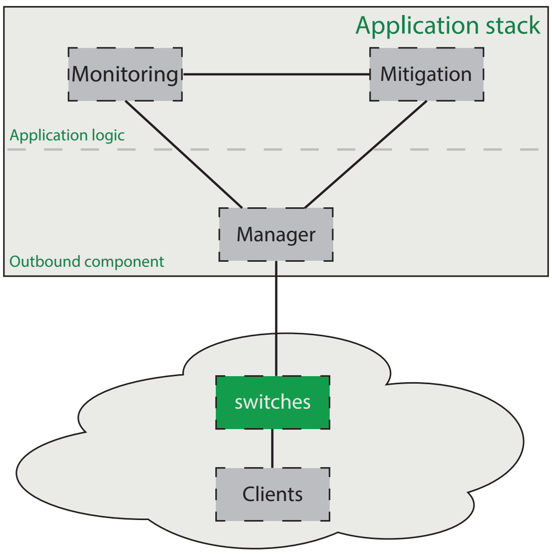

Figure 4.1.: Application stack concept

Responsibilities are split into three base parts. The manager actively communicates on its southbound interface with switches, which provide the SDN network to clients, and are considered the outbound components. Components executing application logic do not communicate with clients directly, only the manager is facing the network side. On its other side, on its northbound interface, the manager provides communication to the application logic, making information actively or passively available to dedicated parts and receiving instructions on how to modify the network. The monitoring components responsibility is to decide whether available data is an anomaly or regular and legitimate traffic. It has to select appropriate tools to analyze data provided by the management component and if an attack is found, an identifiable alert has to be sent to the mitigation component. In summary, its goal is detection. Mitigation reacts upon a received alert with the goal of providing a mitigation strategy. Matching reactions have to be available for different alerts and the actions have to be communicated to the managing component for it to implement changes in the network.

A fourth component can be visualization. This part is not necessary for a working application stack, but it can be used to get insight into the running system thus providing easier use and a smaller feedback loop for human testing. An overview of components and responsibilities can be seen in table 4.1.

Component	Responsibility
Monitoring	Analyze data to generate alerts if necessary
Mitigation	Decide upon a received alert how to mitigate an attack
Manager	Provide a single access point to the network
Visualization	Give insight into events generated by the system

Table 4.1.: Responsibilities of application stack components

Combining all components achieves the primary goal of detecting and mitigating attacks in the SDN environment. Leveraging the three basic concepts keeps components in loose coupling and high cohesion [SMC74], ensuring distributability and portability.

4.2. Information flow

As stated in section 4.1, communication paths and therefore information flow is a vital part of the application. In a large network environment, the sheer amount of information possibly available requires consideration on how to deal with it. In this limited test environment, data flow management is easier compared to a large one. Figure 4.2 depicts which paths information takes to enter the system and how it moves inside the system.

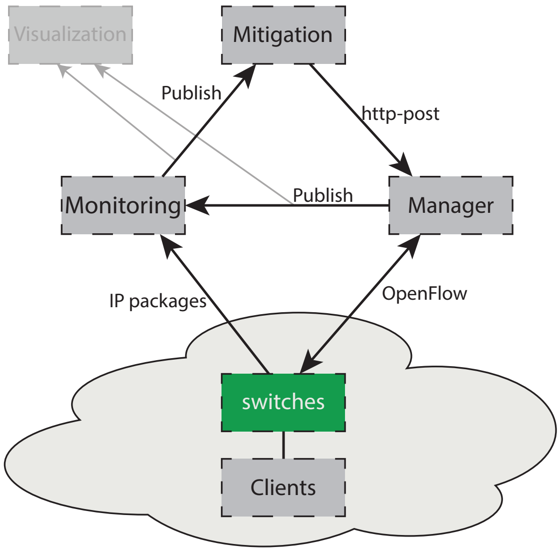

Figure 4.2.: Information flow through the application stack

4.3. Implementation overview

Information flowing into the application stack is always initiated by the manager. This can be statistical information pulled from a switch or IP packet forwarded by one through the network. There are two types of data presented to the application stack. In the first case, IP packets are sent directly to the monitoring component being forwarded by switches. Second, statistical information is gathered by the management component and relayed to the monitoring instance. After data entered the application stack, monitoring decides whether to alert on available information or not to do so. If an alert is generated, it will be received by the mitigation component. This instance will decide on whether the attack is mitigatable and if so, it will inform the managing component about what to do, in order to achieve mitigation. The manager will then install flows according to commands by the mitigation client to appropriate switches. The logical flow of information and the transportation techniques are depicted in figure 4.2.

4.3.1. Network design

The design is chosen with a production network in mind, but aiming for simplicity and ease of use while testing different attacks. Therefore at least two switches are necessary to test how multiple network entry points, with a number of clients attached, behave. Although in a production environment there won't necessarily be a second network available only to connect the monitoring and mitigation stack, this can be achieved by different techniques like VLAN and Quality-of-Service controls similar to a non-SDN -network. In this setup an ideal configuration can be chosen due to no limits concerning available hardware switches and physical paths. Therefore a complete network separation in two different segments is implemented, providing a public network to connect all clients to and a private network providing connectivity to the monitoring and mitigation stack. Though it may be more

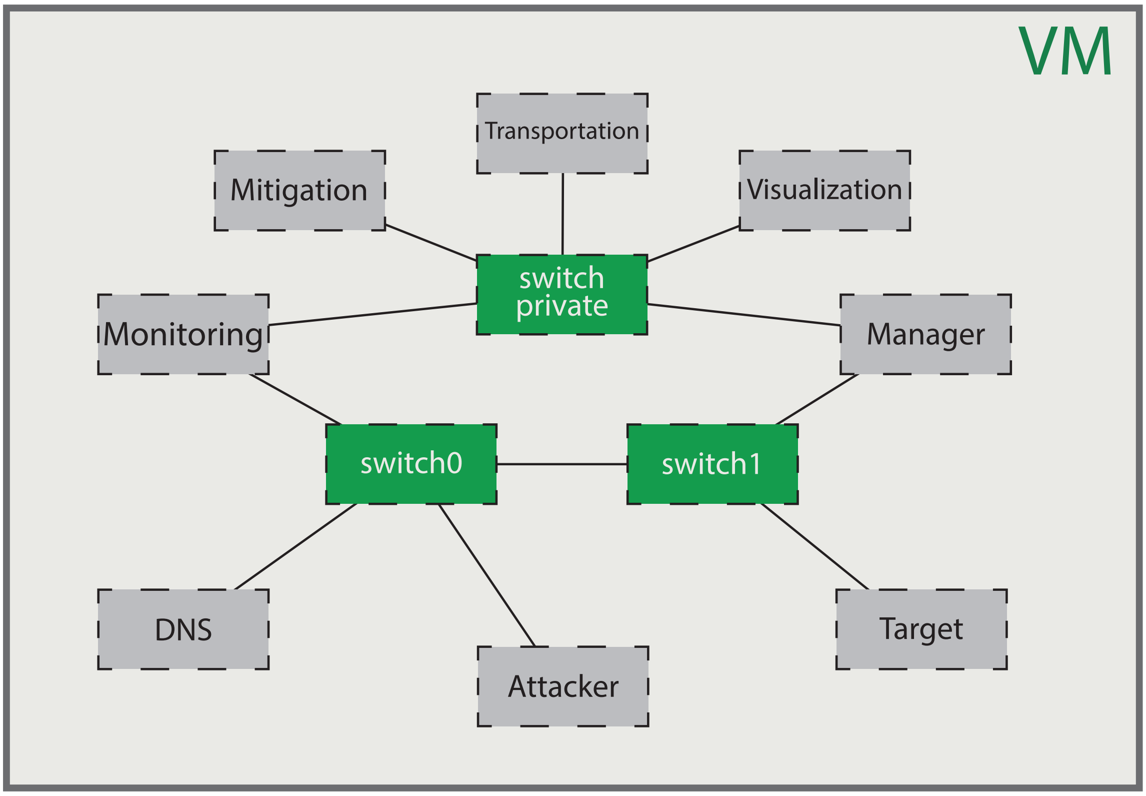

Figure 4.3.: Test-setup of the virtual network

difficult, network separation between the public network and the monitoring and mitigation stack is advisable to minimize the impact in case a system is under attack or compromised.

4.3.2. Monitoring

To analyze any event occurring within the client network, all necessary information gets forwarded to a monitoring client. The same can then decide to alert if specific conditions are met to inform a mitigation client. In this setup, two services are working inside the component. The first one is Suricata (4.4.7) as an IDS. The other one is the Statshandler(4.5.1). More details about monitoring internals are described in chapter 5.

4.3.3. Mitigation

After an attack is discovered by the monitoring component, mitigation can communicate with the manager, trying to mitigate the effects of an attack. Running inside the component are two instances of the Alerthandler(4.5.2), each one a counterpart to the services running in monitoring. This is explained in detail inside chapter 6.

4.3.4. Manager

This component provides an interface to the network. Running Ryu (4.4.5) as an SDN controller, it contains the network application. Responsible for sending instructions to switches, receiving statistics and requests from them on one hand, it is providing a Hypertext Transfer Protocol (HTTP) interface for mitigation to control the network as needed on the other.

4.3.5. Visualization

On top of that, a visualization component provides almost real time insights to events happening within the network. An Elasticsearch-Logstash-Kibana-Grafana-stack(4.4.9) allows for collection, detailed inspection and visualization of occurring events of almost any type and source. In contrast to monitoring and mitigation, which are absolutely necessary to system functionality, visualization is purely for the ease of use and a faster feedback cycle while performing tests in this system. It can also be used to see and notice events, making it easy for a human viewer to correlate between events on different systems. Especially trying to identify the characteristics to monitor while testing an attack and rechecking if a mitigation strategy works is much faster, if data is presented almost instantaneously in graph form.

4.3.6. Component communications

Communication between these components happens in two different ways. First, Redis(4.4.8) enables via its publish/subscribe model high speed data exchange between a single data source and multiple recipients. This is used to communicate between the monitoring, mitigation and visualization component. As a format to exchange data, Extensible Event Format (EVE) is used. The format is natively supported by Suricata(4.4.7), and its JavaScript Object Notation (JSON) base can be easily parsed. Second, a REST-full API is used to instruct the network operating system what to do in case of a mitigatable alert. This ties in with the already provided API and capabilities of Ryu.

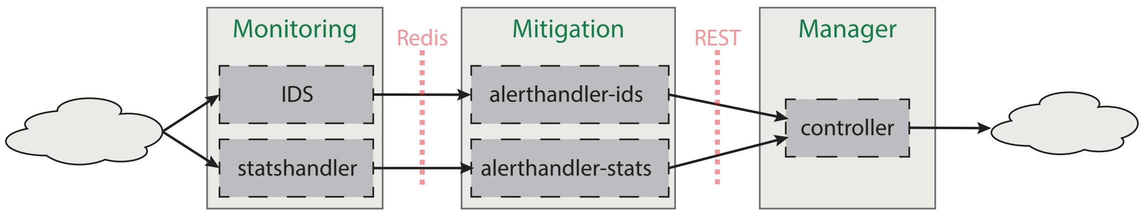

Figure 4.4.: Flow of alert messages

4.4. External software components

The following components are used in conjunction to form the VM setup of this thesis on which all tests will be executed. As many components as possible are open source and all can be used without license fees.

4.4.1. Virtual Box

To ensure encapsulation and portability of the whole test environment, all tests are performed within a Virtual Machine. Enabling simulation of an arbitrary number of devices, multiple topologies can be tested inside the VM. Contained are LXCs, switches (OpenVSwitch (OVS)) and connections between containers via virtual network adapters. To support a broad range of host platforms, VirtualBox v. 5.0.26 is chosen as a platform independent virtualization provider. The base operating system is Ubuntu 16.04 LTS (64bit), both for the virtual machine and the LXC clients. It is installed automatically.

4.4.2. Linux Container

To simulate multiple network clients within a single machine, LXCs are used. They get provisioned via Vagrant and connected to OVSs with virtual network adapters. This enables limiting client resources like available CPUs and RAM via Linux cgroups. Limiting CPU usage is done in the test setup by constraining a container to one CPU. This works also on machines with fewer than 9 cores despite having 9 LXCs. Although it is not possible to truly separate all running processes due to the nature of a single host machine, separation into one container per functionality allows for the best control possible and follows a micro service approach.

4.4.3. Vagrant

To achieve repeatable and reliable tests, an automated setup process for the test environment is necessary. Therefore, Vagrant (1.8.4) is used to automatically provision all virtual machines and containers. The process handles setup of the VM, all LXCs within the machine and connection of all parts to a virtual network. During this setup process, different base images are downloaded from multiple repositories, so Internet access is mandatory. This implies that all containers are connected to an outside network during their setup. This connection can be shut down to ensure network encapsulation and to guarantee all packets sent during the tests will stay within the virtualized network, even if a configuration error is present. Additionally it is possible to commit the infrastructure as code to a version control system and use all advantages like branching and resetting to a previous commit.

4.4.4. OpenVSwitch

OpenVSwitch (v. 2.5.0) is used to simulate OpenFlow capable network switches. Every client is connected to a virtual switch during the setup process. The Networking Operating System is running as a client to control every switch.

4.4.5. Ryu

The Networking Operating System (NOS) controlling all OpenFlow switches is Ryu. It is fully open source and provides a well defined API for communication with the switches in Python. Since Ryu, in the version used for this project (4.5), supports the Open Flow Protocol up to version 1.5, it is the most suitable choice. In the configuration provided by the setup process, Ryu controls the two client-facing switches and provides basic layer 2 network functionality by installing appropriate flows. Additionally, some are installed to duplicate every packet entering the network and send it to a monitoring instance for further analysis.

4.4.6. Bind9

Bind9 is used as a Domain Name System (DNS) server. It runs inside the DNS container and provides its service to all clients connected to the public switches, as known from regular environments.

4.4.7. Suricata

Suricata is used as a packet analyzer and Intrusion Detection System (IDS). It runs as a system service inside the monitoring component and listens on a network interface for incoming IP traffic. It is capable of analyzing different aspects of an IP packet, like it's origin, a TCP/UDP port or its contents. Based on the analyzed content, it can generate alerts if predefined criteria are met. It is manually compiled and not installed from standard packages as they don't support the output of alert information into Redis at the time of this thesis. It was chosen due to its similarities to the widely known IDS Snort and its extended capabilities compared to the same. More details about how Suricata is used in this setup are available in subsection 5.4.1. More information about Suricata itself can be found in its user guide [Her+16].

4.4.8. Redis

> Redis is an open source (BSD licensed), in-memory data structure store used as database, cache and message broker. - [JM16]

As this setup does not only provide monitoring and mitigation functionality, additional message distributing problems are introduced. Namely, both visualization and mitigation need to receive all messages. This raises the problem of a single source and multiple destinations for a given message. Solving this is the publish/subscribe model provided by Redis. It enables a Redis client to publish messages to a channel without having to deal with the responsibility of distributing it to all subscribed clients.

The system is configured to forward all messages in memory. This implicitly means, none are stored on disk by Redis and there is no replay capability. This is not needed in this test setup, but may be necessary in a more complex environment.

4.4.9. ELKG-Stack

This component stack consists of Elasticsearch, Logstash, Kibana and Grafana. Logstash is the first component in line, receiving input messages, formatting and storing them in Elasticsearch. Though there is no capability of doing this in Redis, all transmitted ones are saved in Elasticsearch. All messages are indexed and made available to query for in Kibana and Grafana. Kibana can be used with its 'discover' tab to view the different fields available within the data. Grafana is configured to set up a dashboard automatically at start up in order to visualize the change and relationships of different parameters in the system. More information about Elasticsearch, Logstash and Kibana can be found at the website of Elasticsearch. Details about Grafana are available at `http://grafana.org/`. Instructions how to use the visualization component can be found in subsection 4.7.3.

4.5. Custom software components

Many tasks in the application stack can be handled by pre-written, third-party software. But some components listed in the following chapter are specifically designed, built and tested for this thesis. All services use Python as their programming language, as it was already used by Ryu and therefore necessary for the controller. To ensure consistency and ease of extensibility, all other components were written in Python 3, too. The communication between components is shown in Figure 4.4.

4.5.1. Statshandler

The Statshandler is an analysis service inside the monitoring component responsible for detecting anomalies in statistical data collected from switches. It is running as a system service and can be controlled as such. Its source code is located at `./VM/lxcs/main/monitoring/statistical`. More detailed information about its functionality is found in the monitoring chapter, subsection 5.4.2.

4.5.2. Alerthandler

The Alerthandler is responsible for receiving alerts sent by the monitoring components and deciding how to mitigate an attack, based on available information. Like Statshandler, it is running as a system service. It runs inside the mitigation container and its source code can be found at `./VM/lxcs/main/mitigation`. Its functions and internals are described in more detail in the mitigation chapter, subsection 6.1.2

4.5.3. Controller

The controller is communicating in two different directions. On its northbound API it enables communication to other systems within the network. In this setup, there are two main communication channels northbound. Redis is used by the controller itself to insert statistical data gathered from switches into the application stack. HTTP is used as a second way of communication by the manager to send instructions to the controller in case an alert is triggered.

Southbound, the controller connects to all switches using the OpenFlow protocol. Every time a switch starts, it will announce itself to the controller and waits for instructions from it in the form of flows on how to handle network traffic. The setup process can be seen in listing 4.1. More information about the concept of a north-/southbound API and how it is used in OpenFlow environments is described in section 2.1.3.

```
  return _rest_command


class SwitchController(ControllerBase):
  _SWITCH_LIST = {}
  _LOGGER = None

  def __init__(self, req, link, data, **config):
    super(SwitchController, self).__init__(req, link, data, **config)
    self.waiters = data['waiters']

  @classmethod
```

Listing 4.1: Register switch

4.5.4. SynFlood

There are multiple tools available capable of executing a Transport Control Protocol (TCP) SYN flood, but there is a common problem to them. They are built to execute a blunt flood and don't provide much control over how the exact procedure of flooding is done or when to stop. A similar tool, Hping3 can flood a target with SYN packets, but is not able to stop after a specific amount of packets in flooding mode. These shortcomings are addressed with this tool, which is capable of executing a precise number of packets sent to a target with or without spoofed source IP. Though being implemented in Python, the tool was not slower in this test setup than the in C implemented Hping3.

4.6. FlowFlood

As flow flooding is, compared to SYN flooding, a fairly new attack, no tool was available to produce packets in exactly the manner necessary to trick the controller into installing flows. This problem is solved by the FlowFlood implementation. Working in a master/slave fashion, the program is executed on two different network clients, establishing a TCP connection and sending a small amount of data via the same. Then, the MAC address of the master is changed and a new TCP exchange performed. Every time this is done, a new flow is installed by the system.

4.7. Usage

The virtual machine can be used to simulate the attacks and their effects live. Therefore, a user and development guide is provided here.

4.7.1. Setup VM

The first step in setting up the testing VM is to ensure VirtualBox is installed. Also, Vagrant and the Vagrant-vbuest plugin have to be installed. Almost every part of the installation process needs to download files, so an Internet connection is mandatory.

After changing the working directory in the used shell to `./VM`, the installation process can be started by issuing the Vagrant native command `vagrant up`. If the shell used during the installation process is capable of displaying color, the output will be colored. As some of the installation process happens inside the VM by another instance of Vagrant, the color coding can be wrong, but in general it is normal to see a lot of green lines, some will be red and any lines marking different sections within the installation will be purple. Red lines not necessarily indicate critical errors. For example, output of the curl command will be marked red, but is not an error. If the `vagrant up` command exits with an error code, a critical exception occurred. Additionally, at the end of the installation process, every service will be checked and displayed as 'running' or 'broken'. The installation process should, after installing Vagrant, VirtualBox and switching to `./VM`, only involve the following two commands:

```
vagrant plugin install Vagrant-vbguest
vagrant up
```

Listing 4.2: Setting up VM

All components should be already set up and run after the installation process completes. This is checked once with a script at the end of the installation process. All necessary connections and services must be marked with 'Internet' or 'Running' and the IP for this installation is displayed. To check live status of components, either logging into the VM and rerunning the test script located `/home/vagrant/lxcs/test-main.sh` or looking at visualization of Grafana is possible.

4.7.2. Controlling the VM

To get shell access to all components, the Vagrant command `vagrant ssh` can be used. This connects the used shell in an ssh equivalent fashion to the Virtual Machine (VM). Inside it, the `sudo` command is available without password authentication and necessary for all further LXC commands.

All standard Linux/Ubuntu 16.04-compatible commands are available. Additionally, TMUX and VIM are installed. To ensure that containers are in fact started, `lxc-ls --fancy` displays a list of running LXCs and their IP addresses. To connect to one of them, the command `lxc-attach -n container-name` is available. When logged into the VM, all Linux/Ubuntu 16.04 commands are available inside all LXCs. Some example use cases from within the VM are shown in listings 4.3 to 4.5.

```
lxc-stop -n monitoring
```

Listing 4.3: Stopping a container

```
lxc-attach -n mitigation
systemctl restart alerthandler-stats
```

Listing 4.4: Restart a service

```
lxc-attach -n monitoring
journalctl -u statshandler -f
```

Listing 4.5: Following the log output of a service

4.7.3. Visualization

To visualize all information produced by the system, two services are in operation. Kibana is the standard visualization software used in many Elasticsearch-Logstash-Kibana installations. Using Kibana, all unique fields are visible from the 'Discover' tab. Kibana is available via port 81 of the VM, login data is 'kibanaadmin' as user and '123' being the password. Grafana is used to visualize multiple graphs. It can be used to see time correlated events and build almost real time insight graphs to display multiple source fields from Elasticsearch. Grafana is available via port 80 with the login 'admin' as user and password.

More information about how to configure and use Grafana can be found in its documentation: `http://docs.grafana.org/`. Equally, usage information about Kibana can be found in Kibanas documentation: `https://www.elastic.co/guide/en/kibana/current/index.html`.

4.7.4. Starting attacks

An attack is always started from within one of the LXC attackers. All scripts starting an attack are located in `/vagrant/attacker/` and can be executed without any parameters to use the predefined targets. As these are simple bash scripts, the target can be changed by modifying the script. For example, to execute a simple DOS attack one would issue the following commands:

```
lxc-attach -n attackone
cd /vagrant/attacker
./synflood.sh
```

Listing 4.6: Starting DOS attack

As long as the command is running, a packet flood is visible in Grafana.

4.7.5. Development guide

Extensions and changes to the project should follow the already available directory structure. All information necessary for a component to offer its service should be located in the components folder. In case of an LXC this would be `./VM/lxcs/container-name/`. To modify the network behavior, changes to the controller can be made in `./VM/lxcs/main/manager`, either directly to controller.py or by creating new modules.

To create or modify attacks, files in `./VM/lxcs/main/attacker` can be added or changed. Changes to monitoring and mitigation would then happen in `./VM/lxcs/main/mitigation` and/or `./VM/lxcs/main/monitoring`.

The development of custom Python modules follows the style guide PEP8 with a line width of 120 characters. Python 3 is preferred over Python 2 as all current modules are written in Python 3.

Bash is the used shell scripting language.

4.8. Advantages and disadvantages

This testing environment was chosen due to its advantages. The most prominent is repeatability. Due to the capabilities of Vagrant, the VM can be set up from scratch again and again with one simple command, providing a fresh install of the test setup every single time. Ensuring the same test environment not only in between test runs, but also across a development team, makes testing repeatable and development easy. Even if something inside the VM stops working, e.g. the virtual hard disk fills up due to too many packet capture files, the whole machine is simply disposable and set up from scratch in a few minutes. Additionally, by specifying architecture in text like files, version control is simply done in git, allowing version controlled architecture and the merging of system features. Using LXCs inside the VM allows for a full network setup while keeping the most separation possible inside a Linux machine. With this container based solution, every single LXC runs like a separate machine, including the possibility used in this setup of running system services via SystemD. Again using Vagrant, to set everything up, makes it easy to spawn new containers. In summary, the VM is as invisible as possible to the test setup and every function is run in a container.

Disadvantages of this setup include an incomplete separation of network components. Though separated, all containers still run on the same hardware sharing the same VM. This leads to the possibility of unwanted influence from one container to another, e.g. in disk IO operations. These will still read and write from and to the same disk. Another disadvantage is a possible scaling issue. This setup is not intended to be used with a lot of clients, and this problem could only be solved to some degree by providing a larger VM host. Nevertheless, solutions to the problem would be splitting the VM onto multiple hosts and connecting them via hardware. Additionally, there are some security concerns discussed in the following section 4.8.1.

4.8.1. Difference to a production environment

As this setup is a testing environment, it is not production ready. The intent in building this setup was to provide ease of use and proof of concepts. Therefore, the network and application design is not transferable to a production environment as there are no physical distance limitations within a VM. Additionally, some security measures found in a production environment are not implemented in this setup. To run a similar stack in production, the setup process and the implemented security measures would have to be adapted to the real network and the security guidelines in place. A minimal set of changes necessary are the following.

The setup files now used to set up all LXCs have to be adjusted for at least another provider, if Vagrant is still used as a setup manager and only the provider changes. After adjusting the IP addresses and domain names of all containers, the shell part of the setup process can be used to repeat the installation on a variety of hosts capable of running Linux/Ubuntu 16.04 and executing a shell script.

All set passwords should be set to secure ones. Neither the VM base machine used as template, nor theLXC base follows any security guidelines other than the defaults provided by the Ubuntu operating system. The `sudo` command can be used without password. Additionally, Vagrant mounts shared volumes to all components for the ease of use of sharing files. These may be replaced by NFS shares if necessary.

There are currently no authentication and authorization processes in place. The communication via HTTP from mitigation to manager is not secured, a switch to Hypertext Transfer Protocol Secure (HTTPS) would be necessary. Interaction with Redis also happens unencrypted and without authorization by all clients. Likewise, the connection between switches and manager is also insecure. Secure and proven communication possibilities are available for all these channels so a production ready and security guideline compliant setup can be built.

An overview of minimum security changes is shown in Table 4.2.

Component	Current Setup	Secure Setup
Redis	Every client with a connection to the secure switch has access to Redis	Use stunnel to encrypt communication[Hab14]
OpenFlow	Plain TCP messages to the controller	Switch to secure mode for all switches and TLS for all communication[Nyg+14]
HTTP communication	Simple HTTP messages between mitigation and manager	Use TLS for all HTTP traffic
Visualization	Plain HTTP and simple passwords	Use TLS and change passwords to secure ones
Identity provider	Vagrant SSH with generated keys and sudo without password	Implement deployment specific security guidelines

Table 4.2.: Example changes to secure setup

5. Monitoring

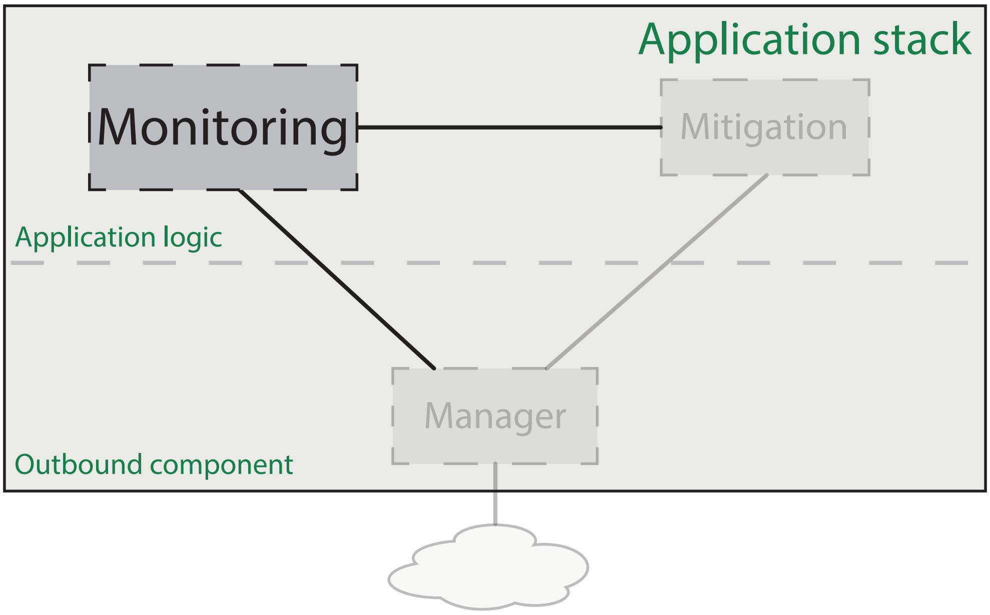

Figure 5.1.: Monitoring in the concept

As a first step in the pipeline to a successful mitigated attack, a working monitoring system is key. All further actions require reliable information and alerts to act upon. At the same time, to keep latency between the start of an attack and mitigation of the same low, the monitoring system has to operate in real time. In contrast, to take the best informed decision possible, as much information as possible needs to get analyzed. This means getting as much relevant information as possible to the component responsible for analyzing it and having all of it analyzed fast enough to keep up with the network.

In itself, the monitoring component has to fulfill three steps. First in line is to gather data from external sources to a decoding component and making it available locally. Then the application logic within monitoring can analyze presented information to decide whether the last step of alerting mitigation shall be performed. An overview to this behavior is given in figure Figure 5.2.

The following sections will give insight into different methods and techniques to perform the steps necessary to generate valid alerts from raw data. First, the concept of the three phases used will be described, to be followed by a description of what is implemented in the VM. Then, the three attacks described in section 2.2 are executed and an analysis of the events is provided.

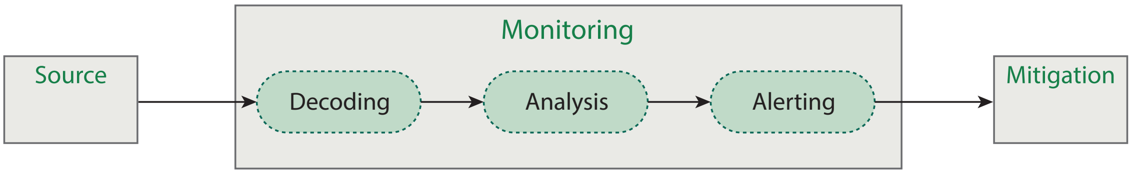

Figure 5.2.: Monitoring concept

5.1. Data Gathering

Regardless of the type of data gathering used or the type of data collected, all data has to arrive at the monitoring component. The following compares three different methods of collecting data.

5.1.1. Packet Capture

Packet capture can catch every packet injected into the network and forwarded to monitoring. After decoding the input, statistics can be computed using signatures in the analysis phase.

To generate alerts in a reliable way, the component has to analyze as many potentially important packets as possible and therefore receive a large number of them. If there is no filtering previously to decoding the packets, this results in the disadvantage of having to deal with all packages available. Though it is possible with SDN to do so, it can be a problem not only for the component itself, but also for other parts of the network involved, to reroute many packages to a single destination. To minimize this problem, the component can be placed at already existing network constraints like routers, firewalls or other forwarding devices.

Physical limitations like bandwidth can cap the amount of packets a component can receive, especially in situations where traffic floods traverse the network (e.g. DoS attacks). A client can have direct influence on how many packets have to be processed by monitoring, so it may be suitable to distribute package capture.

Another issue emerging from the number of packets captured in combination with how complex and resource intensive the analyzing logic is, is that performance can be a problem. Due to the constraint of having to keep up with the network traffic, thorough analysis of every packet may result in the component not being able to capture all following packets, resulting in missed information. An advantage of having this much source material in its raw form, namely whole packets, is the flexibility to decide in the analysis phase on any signatures detectable inside the packet, including payload of any kind. This enables picking a select number of packet features independent of other ones like e.g. alerting on payload content independent from source IP address. A sample of possibilities can be seen in the Suricata documentation [Her+16, 4. Suricata Rules] In summary, packet capture provides a lot of raw information to the next phase. This is an advantage and a disadvantage at the same time.

5.1.2. Network Metadata

Network metadata is all data that can be retrieved by gathering information from the network infrastructure itself. In an SDN environment this is possible using the OpenFlow protocol.

Every switch can gathers data about its ports and installed flows. Available counters can be seen in the OpenFlow specifications [Nyg+14, p. 32]. This information can be relayed via the management component to monitoring and processed to detect anomalies.

A fact to consider when setting up network monitoring via metadata is, that information is implicitly grouped and filtered by the flows installed. As in OpenFlow a packet is assigned to a flow via the its match fields, these determine also the granularity of monitoring information available. In OpenFlow v. 1.5 these can match many header fields including an Ethernet frame, all available are listed in the OpenFlow specifications [Nyg+14, pp. 77-78]. As a result of the assignment to a flow, high coupling is present between routing information on one side and monitoring data on the other. Additionally, the matching capabilities are limited ,the highest OSI layer [ISO, p. 28] protocol fields matchable according to the specification are TCP/UDP fields on layer 4, meaning it is not possible to monitor anything contained within TCP/UDP packets or non standard header fields used by other protocols.

An advantage of network metadata monitoring is the offloading of a part of the decoding process. In such an environment, the local load on the monitoring component is reduced, as the network already provides information preprocessed and the component itself can focus mainly on analysis of the presented information. Tying into that advantage is a lower network load, as fewer packets containing just the information provided by switches, have to be forwarded to monitoring. Additionally, control over how many packets are sent is with the network provider, as polling intervals and the number of packets gathered and relayed to monitoring can be configured by the provider and cannot be influenced directly by a client. As a summary, monitoring network metadata is receiving statistical data from the network itself and deciding based on that data if an alert is necessary.

5.1.3. Active Client Monitoring

Monitoring can also happen on the client itself. This requires control over the same and additional software in place to forward data to the monitoring component. All information about a single client is thereby implicitly grouped together.

With this method, some of the monitoring workload can be spread across the clients, so that only important information has to be forwarded to the monitoring component. Detecting anomalies involving only a single client could happen within the same, not crossing the network boundary and without any dependencies to a network. To monitor distributed information, a solution is to send it data to a centralized monitoring component.

This can happen independently from the network solution chosen, whether it is a regular network or an SDN approach, as it uses the network only as the transport media and does not rely on any other features. A drawback of active monitoring is, that in some cases, not every network component is accessible and it is not possible to install additional software on it. Should a system be compromised, the information coming from this client may not be reliable and report inaccurate information to the monitoring component.

In summary, monitoring via an active client setup is a distributed approach with the possibility to leverage processing power of clients to analyze only preprocessed data centrally.

5.1.4. Comparison

Each method of data collection has its advantages and drawbacks, which shall be compared in a short summary in Table 5.1 .

	Pro	Contra
Packet capture	Fine grained overview of all data in the network	Costly to analyze lots of data
Network Metadata	Overview about data flowing through the network	Only minimal data content analysis possible
Active Client Monitoring	All information from a specific source combined	Every client has to be setup separately

Table 5.1.: Comparison of monitoring methods

5.2. Data Analysis

The decision whether to alert on a set of given data or not do so is the main responsibility and challenge of the analysis phase. Independent of any network technology, this module has to compare the presented data to values the data should have. The data is already decoded and there are multiple ways to compute a range of data in which case to generate an alert. The goal of this phase has to be to generate alerts for as many happening attacks as possible while not issuing them for valid traffic.

5.2.1. Limit

A rather simple method is a fixed limit of x, converting to a range from $\{0 \ldots x\}$ or an explicit set range $\{x \ldots y\}$. For a number of values, a fixed range may be enough. A number of monitoring tools widely used like Nagios or Icinga provide this functionality [Tea00]. For example, if in a known network the number of IP addresses originating from a switch port is monitored, a range of zero to one IP address per port can be a reasonable limit, if this is a port known to have only one client attached to it.

5.2.2. Standard Deviation

A slightly more complex method is computing an allowed range for the next value. This can be done by computing the standard deviation over a number of last received data points n, and generate an alert, as soon as a new value is outside of the range computed by the equation from Figure 5.3. However, the technique can not work without any data or with rapidly and suddenly changing values. The data rate on a main connection link, forwarding a lot of packets, could be monitored this way. If an almost constant rate with only marginal deviations is expected, then this method can give a lower and upper bound on which to alert on.

$$
\begin{aligned}
top &= x + 2 * \sigma_n && (5.1)\\
bottom &= x - 2 * \sigma_n && (5.2)
\end{aligned}
$$

x: Last measured value, σ: Standard deviation, n:Number of values to calculate σ over

Figure 5.3.: Range by standard deviation

5.2.3. Triple Exponential Smoothing

A third possible way is to use Triple Exponential Smoothing (TES) [Fil+13, p. 6.4.3.5]. This method combines three smoothing equations as seen in Figure 5.4 to compute a forecast of values.

$$S_t = \alpha \frac{y_t}{I_{t-L}} + (1-\alpha)(S_{t-1} + b_{t-1}) \quad \text{OVERALL SMOOTHING} \tag{5.3}$$

$$b_t = \gamma(S_t - S_{t-1}) + (1-\gamma)b_{t-1} \quad \text{TREND SMOOTHING} \tag{5.4}$$

$$I_t = \beta \frac{y_t}{S_t} + (1-\beta)I_{t-L} \quad \text{SEASONAL SMOOTHING} \tag{5.5}$$

$$F_{t+m} = (S_t + mb_t)I_{t-L+m} \quad \text{FORECAST} \tag{5.6}$$

y: Observation
α: Estimated overall smoothing constant
S: Smoothed observation
β: Est. trend smoothing constant
b: Trend factor
γ: Est. seasonal smoothing constant
I: Seasonal index
t: Index denoting time period
F: Forecast at m periods ahead

Figure 5.4.: TES. Based on [Fil+13][6.4.3.5]

A range of accepted values can then be computed in multiple ways, a simpler one would be e.g. by allowing a fixed percentage deviation as shown in Figure 5.5.

$$top = F_{t+1} * 1,1 \tag{5.7}$$

$$bottom = F_{t+1} * 0,9 \tag{5.8}$$

F: forecast at time period t

Figure 5.5.: Range by fixed percentage

The advantage of this method is that it encompasses multiple ways a value could change over time and adapts to those changes. Then again, this is the most resource intensive method presented here. A second disadvantage is, that in order to compute an initial trend required for the forecast, at least one complete season of data has to be available. Depending upon a season length, this may be quite an amount of data necessary.

5.2.4. Comparison

As each data gathering method, every analyzing technique has its advantages and disadvantages too. Table 5.2 shows a summarized comparison between the presented possibilities.

	Pro	Contra
Limit	Simple to set, fast to implement, few resources required	Rigid, unresponsive to change
Standard Deviation	Adaptive, few prerequisites	Only suitable for slowly changing values
TES	Most adaptive and comprehensive	One season of data necessary, most computing resources used

Table 5.2.: Comparison of analyzing methods

As some methods could counteract disadvantages of others, a combination of approaches is possible. For example, when using TES to monitor the bandwidth used on a connection, it may be feasible to set a maximum as a first check to alert on. The limit could alert if used capacity is higher than 95% of the link capacity and save computing resources in this case, while, if this limit is not hit, TES could still check for an acceptable bandwidth use.

5.3. Alerting

As the last step in this component, the alerting phase is responsible for communicating results. As there is only communication happening in the case of an alert, it has to fulfill the following criteria.

- Reliability
- Speed
- Lucidity

As the communication channel for alerting is empty unless there is an alert, it is crucial to receive every sent packet and not miss a single one. A reliable and proven protocol has to be chosen to ensure a solid communication path.

To keep latency low, speed is an important part too. Especially in a situation where an attack is already detected, and presumably still active, the information about this attack has to be received by the mitigation component as fast as possible. This is true for all kind of attacks, but notably in any flooding attack a malicious network client uses every chance it has got to insert packets.

A last but nonetheless important criteria is lucidity. This encompasses, that a message received by another component has to be identifiable unambiguously, must be parsable fast and contain every bit of information necessary without being overloaded. The recipient should be able to decode a message without the necessity to compute additional information in an intermediate stage. With the only input being the alert message, the recipients should be able to use the presented input in combination with its own information.

Existing software, like e.g Prelude OSS, uses Intrusion Detection Message Exchange Format (IDMEF) [Deb+07] to communicate alerting information [Sys15]. This format leverages

Extensible Markup Language (XML) via TCP as its transportation provider to communicate a range of information.

5.4. Implementation

The VM described in chapter 4 implements a selection of the depicted functionality. This chapter explains in detail the implementation concerning the monitoring component, its Linux Container (LXC) and how it interacts with its surroundings.

To keep a separation of concerns, both monitoring channels are used as independent system services and can operate without any dependency to each other. It is possible, without changes to the application logic itself, to extrude one of the services into another location and reroute its traffic without affecting the other one. A depiction of the implementation internals of the monitoring component is shown in Figure 5.6.

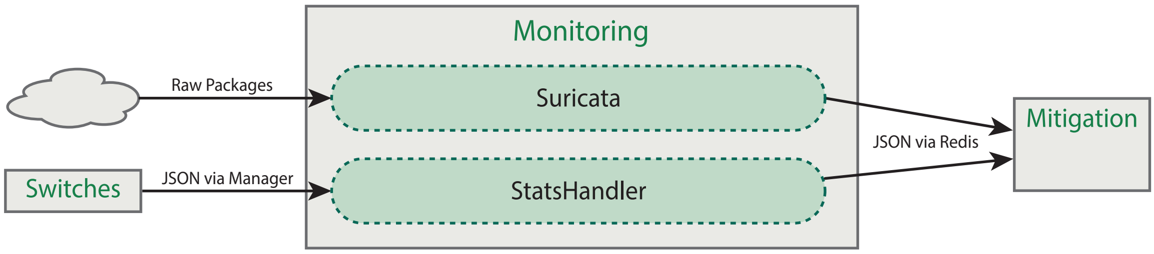

Figure 5.6.: Monitoring implementation

5.4.1. Packet Capture

Suricata is used to implement packet capture and analysis. It is used as an IDS, reading every packet arriving at the Network Interface Card (NIC) of the LXC. All three steps depicted in Figure 5.2 are handled by Suricata.

To get packets to the LXC, every switch in the network is, at startup, instructed to duplicate every packet and send one to its destination and the other to the monitoring instance. The subsequent decoding process is done from raw packets, making data available for the analysis phase. Suricata as a signature based IDS uses rules to decide when to alert. An example rule is shown in Listing 5.1.

```
alert udp 192.168.127.15 any -> $HOME_NET 1 (msg:"Malicious␣Host"; GID:1;
      sid:60000003; rev:002; flow: stateless;threshold: type both, track by_src,
      count 500, seconds 10;)
```

Listing 5.1: Example Rule

The rule generates an alert, if the host 192.168.127.15 sends more than 500 packets within 10 seconds from any port to any host in the 'HOME_NET' at port 1. After the generation of an alert, it is published in JSON format to the Redis instance making it available for consumption by other components.

This implementation was chosen due to its simplicity. Suricata is able to handle all phases of the monitoring component. It is easy to configure and proves itself fast enough in the attack analysis section 5.5. The output format JSON was chosen due to its wide spread use.

The output path of Redis enables this test setup to send the alert messages to more than one recipient and interfaces with the statistics Suricata is configured to produce. A drawback to this way of implementing the forwarding is, that internally the packet number is doubled. An attacker therefore can generate double the pressure to the network, although packets have different target clients. Another problem is that IP spoofing is not detected. If the attacker uses a spoofed IP, the monitoring instance generates an alert containing this false IP. In this implementation, Suricata has no information about which OVS port a packet is sourced from.

5.4.2. Network Metadata

OVS supports the collection of metadata as specified in the OpenFlow specifications. All client network switches are probed by the manager for statistics, and they deliver back values about every port and the flows installed. A sample excerpt from a statistics message as sent by a switch can be seen in Listing 5.2.

```
"1": {
 "rx_packets": 1909,
 "tx_bytes": 100862,
 "tx_errors": 0,
 "properties": [
  {
   "OFPPortStatsPropEthernet": {
    "collisions": 0,
    "rx_crc_err": 0,
    "type": 0,
    "rx_frame_err": 0,
    "rx_over_err": 0,
    "length": 40
   }
  }
 ],
 "tx_packets_delta": 0,
 "rx_dropped": 0,
 "rx_errors_delta": 0,
 "length": 120,
 "tx_bytes_delta": 0,
 "tx_dropped_delta": 0,
 "duration_sec": 500,
 "tx_errors_delta": 0,
 "rx_bytes_delta": 0,
 "duration_nsec": 212000000,
 "rx_dropped_delta": 0,
 "rx_errors": 0,
 "rx_packets_delta": 0,
 "rx_bytes": 5999847,
 "tx_packets": 1644,
 "tx_dropped": 0
},
```

Listing 5.2: Sample port statistics

In case of this implementation the controller additionally calculates delta-values to previous fetched statistics as seen in lines 17 to 24 at Listing 5.3. As these values are already available inside the NOS, it is easier to calculate them within the controller and then pass them on to the monitoring instance to easily analyze differences in historical values.

```
          # Default value for mitigation flows
          table_id = 0
        if REST_COOKIE in data:
          cookie = data[REST_COOKIE]
        else:
          cookie = 0
        if REST_COOKIEMASK in data:
          cookie_mask = data[REST_COOKIEMASK]
        else:
          cookie_mask = 0
        if REST_IDLETIMEOUT in data:
          idle_timeout = data[REST_IDLETIMEOUT]
        else:
          idle_timeout = 0
        if REST_HARDTIMEOUT in data:
          hard_timeout = data[REST_HARDTIMEOUT]
        else:
          hard_timeout = 0
        if cmd == 'OFPFC_DELETE':
          self.ofctl.delete_flow(priority=priority, table_id=table_id,
  match=match, command=command_obj,
                       cookie=cookie, cookie_mask=cookie_mask)
        else:
          self.ofctl.add_flow(priority=priority, table_id=table_id, match=match,
  action_list=action_list,
                      hard_timeout=hard_timeout, idle_timeout=idle_timeout,
  command=command_obj,
                      cookie=cookie, cookie_mask=cookie_mask)
        self.logger.info(
          "Added␣mitigation␣flow:␣{}{}{}{}{}{}".format(priority, table_id, match,
  action_list,
                                  hard_timeout, idle_timeout), extra=self.sw_id)
        details = 'Performed␣%s' % cmd
```

Listing 5.3: Statistics gathering

The analysis phase is then executed by the Statshandler running within the monitoring LXC. There, thresholds are implemented that get tested against received values. If a value surpasses a limit, an alert is published. This is done via Redis in JSON format again for the same reasons mentioned before. As there was no software available solving this exact problem, the implementation was done manually. This allowed for the most flexibility and a seamless integration into the application stack. A second drawback is a time issue. The Statshandler component does only get data every 5 seconds. This means, in a worst case, the delay between the start of an attack and its detection is at least 10 seconds. Compared to an IDS, which gets packets as fast as the network can forward them, this is slower.

5.5. Attack Analysis

The following sections will analyze three different attacks and for each one multiple ways to detect the it. Each of the methods will have its advantages and drawbacks highlighted to be able to compare them against each other.

All attacks implemented are performed on a freshly set up VM. Inside of it, they will be launched and monitoring data will be gathered. This data is used as a source for shown figures and to compare methods.

5.5.1. SYN Flood

Starting simple, the first example looked at is a DoS attack originating from a single host. After connecting to the LXC 'attackone', it can be started with the command shown in Listing 5.4.

```
/vagrant/attacker/ping-flood/SynFlood.py -s 10.10.10.3 -t 10.10.10.4 -n 5000000
    -p r -P 80
```

Listing 5.4: Starting a DoS attack

Suricata

Figure 5.7 shows an unmitigated dos attack originating from a single host machine, attackone in the test setup. The attack was performed with 5 000 000 packets sent in 72.30 seconds ($\approx$ 69 000*pps*) by the attacker. In this test, the mitigation component was shut down, to allow the full attack to spread through the network.

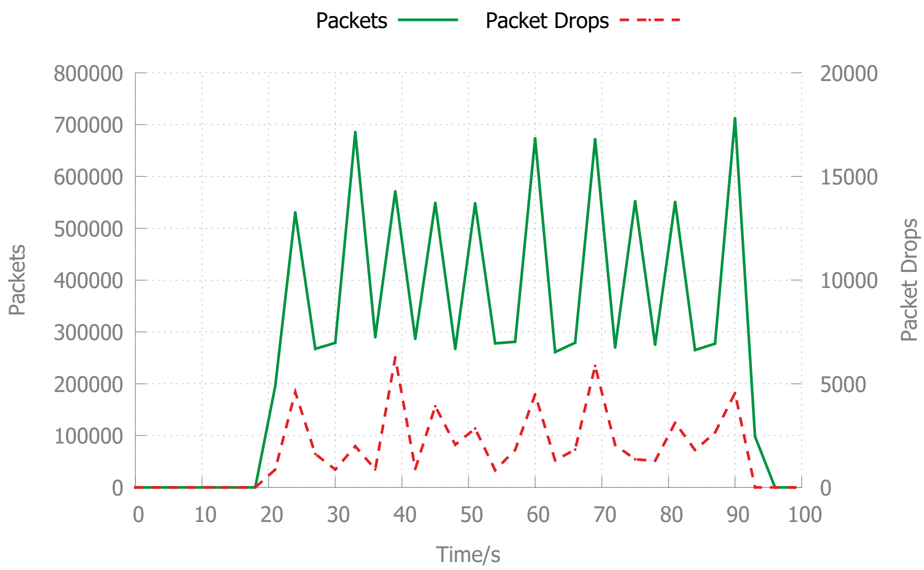

Figure 5.7.: DoS attack without any mitigation

If the attack stays unmitigated, Suricata starts loosing packets in its analysis phase. The monitoring instance is flooded with packets and starts dropping them in order to analyze new ones as can be seen in Figure 5.7. Detecting a SYN flood is possible using Suricata. The goal in catching the attack is, to identify the traffic and host from which it originates. The rule used to achieve this is shown in Listing 5.5. In summary, this rule generates an alert, if the following conditions are met: There where more than 500 packets from the same source IP address within the last 10s having any source and destination IP, any TCP ports and their TCP SYN flag set. In case all conditions are met, an alert with the id 10 000 003 is generated.

```
alert tcp $HOME_NET any -> $HOME_NET any (flags: S; msg:"Syn␣flood␣detected";
    GID:1; sid:10000003; rev:001; flow: stateless;threshold: type both, track
    by_src, count 500, seconds 10;)
```

Listing 5.5: DoS Rule attack

With a rule like in Listing 5.5, it is possible to pin a single attacker precisely. The alert generated contains every identifying value of the attack. These are source ip, destination ip, TCP source port and TCP destination port, in Listing 6.2, a full alert message can be seen. The method using a simple limit is sufficient for this set up. Regular clients pulling a test page from the web server running on the 'target' LXC do not surpass this limit. Though it is not adaptive, it is enough as the regular traffic is known.

Statshandler

A second method in detecting a DoS attack is using statistical analysis. The Statshandler service within the instance is monitoring all switch ports. This allows the detection of an unusual behavior of client ports, but gives no insight in what traffic is sent. As seen in Listing 5.2, a number of different metrics are available, but none of them contain any information about what was transmitted other than the number of valid Ethernet frames. To get this information, a flow based analysis has to be used. The flow has to match the attack characteristics as close as possible. It would resemble the Suricata rule loosely without containing information about the rate of packets. A possible implementation would be, to leverage multiple tables in the OVSs. Then, the first table can contain matching rules, some catching every SYN packet per port and one, matching every other packet. All rules forward the packets to table two, which outputs all packets to their destinations. This setup would allow counting all TCP SYN packets within one flow per port. An overview to this setup is shown in Figure 5.8.

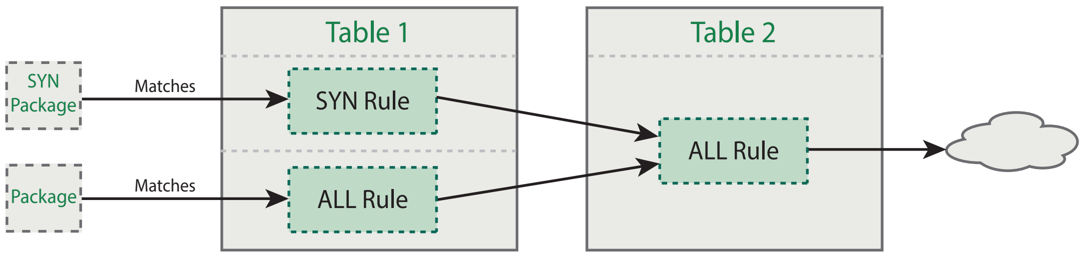

Figure 5.8.: Flow of alert messages

With this information present, the statistic analyzer can compare the number of SYN packets on a single port to its logic and, if necessary, generate an alert containing the source OVS port and the information that the attack is a SYN flood. The alert therefore does not contain every information about the attack. TCP source, TCP destination port, and both source IP and destination IP are missing. This is an advantage compared to a solution monitoring ports only, as the link between a heightened packet count and TCP SYN packets is present. Compared to using Suricata, this can not detect the target IP or the TCP destination port used. These informations are used by mitigation in subsection 6.2.2 to generate an alert.

5.5.2. Distributed Syn Flood

A distribution across multiple hosts brings more challenges to the network and to the monitoring component. The main difference for catching such a distributed attack is the missing characteristics of a fixed source IP address and a single OVS source port.

Suricata

In the setup present, Suricata is fast enough to issue multiple alerts in the same way as done for a single source SYN flood, as there are only a few hosts. But as soon as these attackers are using spoofed IPs, this is equal to a lot more hosts, at least in the informations available to Suricata. Additionally, for the IDS to detect the attack, a threshold of 500 packets per 10 seconds has to be surpassed. Using the rule shown in Listing 5.5, this may never be the case if attackers never surpasses the limit with one of the spoofed IPs. Additionally, this generates a lot of load on Suricata, as it tries to count SYN packets issued for every IP individually. To circumvent this problem, the rule shown in Listing 5.6 is used. The two changes compared to Listing 5.5 are 'sid' and 'track by_dst'. This instructs Suricata to count sent SYN packets for every IP destination instead of every IP source address.

```
alert tcp $HOME_NET any -> $HOME_NET any (flags: S; msg:"Syn flood detected";
    GID:1; sid:10000004; rev:001; flow: stateless;threshold: type both, track
    by_dst, count 500, seconds 10;)
```

Listing 5.6: DDoS Suricata Rule

The effects on Central Processing Unit (CPU) usage and an unmitigated attack using spoofed IPs is shown in Figure 5.9. In both cases, using `track by_src` and `track by_dst`, the CPU usage increases to 100%, shifted in time as Suricata processes packets. As the IDS reaches full load, it starts to drop packets and is not able to keep up with the flood of information. During the process, packet count drops down to 0, as the monitoring instance cannot handle the load and produces inaccurate packet counts. After the attack ends, CPU usage levels drop back to normal again.

The important difference is that when using `track by_dst`, Suricata is fast enough to detect the attack and issue multiple alerts before it starts to drop packets and run out of computing resources. In case of this implementation, Suricata therefore is fast enough to detect the attack multiple times. This can be seen in Figure 5.10.

In summary, Suricata is capable of detecting the attack, but cannot keep up with it for a large amount of time.

Statshandler

One way to solve the problems of Suricata is using the Statshandler component. As it receives its information from the OVS infrastructure, it doesn't have to decode all packets first. Presented with a statistics summary, the decoding process is limited to parsing the JSON message and it can start its analysis phase right away. In Figure 5.11 an attack executed with the same parameters as in Figure 5.9 is shown, but without Suricata and the mitigation stage running.

This shows, that Statshandler can generate alerts more reliable than Suricata in case of a distributed SYN flood attack with spoofed IPs. The instance is not involved with the packets transmitted directly and can act through the attack constantly. It is slower though,

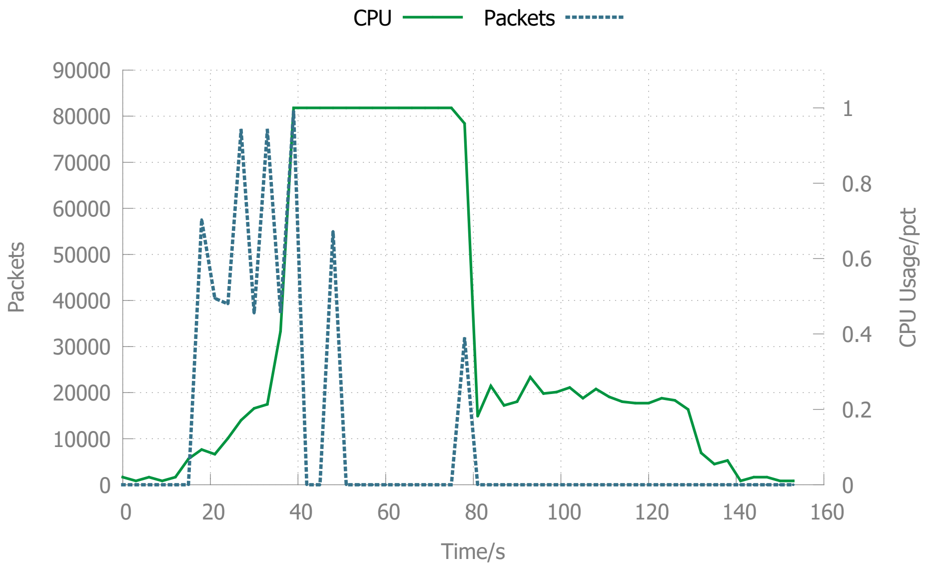

Figure 5.9.: SYN Flood with `track by_dst`

as it has to wait for switch statistics to arrive while Suricata is able to react as soon as viable packets arrive.

The implementation uses a similar fixed limit as Suricata does. An alert is generated, if more than 500 packets are received and transmitted on a port within the 5 seconds time window after which new statistics are received from the controller. To detect ports that insert many packets into the network, this limit is sufficient. Even if not only IPs are spoofed and there truly are multiple clients inserting packets, the Statshandler relays more than one alert with a limit-surpassing port.

In this use case enough, it bears some problems. As the Statshandler can not differentiate between the type of packets, it is possible that if multiple ports send many packets, the Statshandler will not only report the ports sending SYN packets, it will also state all other ones.

Combination

To achieve the goal of detecting attacking hosts precisely, the two methods stated earlier can be combined. If Suricata and the Statshandler run in parallel, information from both services is available. Combining the destination IP, destination TCP port and set TCP SYN flag from Suricata and source OVS port from Statshandler narrows the packets identified as part of the attack further down. It still is possible that legitimate traffic is blocked if it is inserted into the network at the same port as used by an attacker and its target is also the attacked host. But communication flowing from a non-involved source, spiking network traffic at a OVS port to to be forwarded to a different destination will not be blocked. Combining both approaches is therefore the best solution, adding up the advantages from both methods.

5.5.3. Flow Table Flooding

As a, in contrast to DoS and DDoS, unique attack to SDN, flow flooding is more challenging than other attacks. The following sections will show that Suricata has its difficulties in

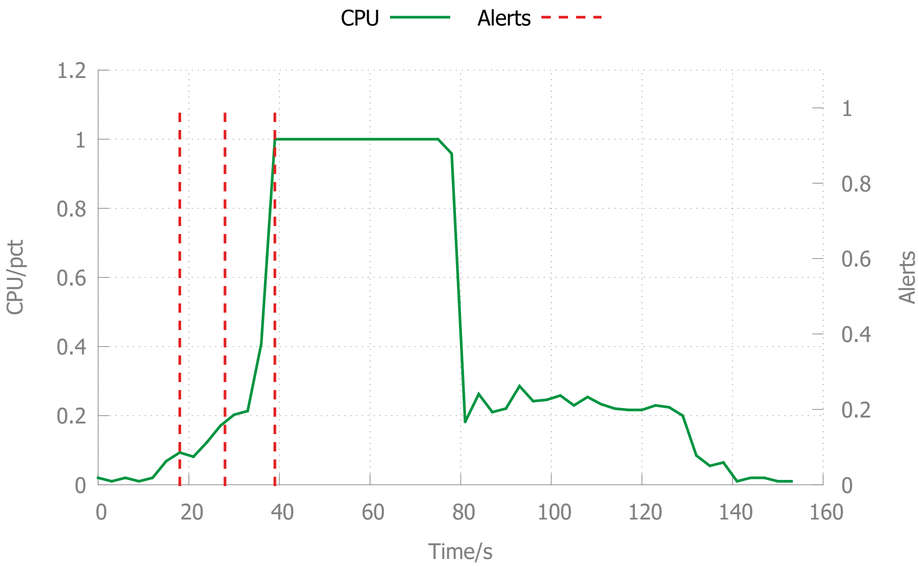

Figure 5.10.: SYN Flood with alerts

dealing with the attack, due to its SDN nature. The Statshandler can work more efficiently due to its capability of interfacing with the information available in the SDN infrastructure.

Suricata

As Suricata can not access the information available in the SDN directly, it can perform two classes of actions. One approach, is to inspect every packet in the network for content that possibly results in more flows. On one hand this depends heavily on the implementation of how flows are generated and on the other hand on how this implementation is exploited. If a common denominator can be found to identify packets generating flows, this characteristic can be searched for and an alert can be generated if suspicious behavior is detected. As in this implementation, the attacker and its partner are sending packets back and forth having the content `ping` and `pong`, Suricata can match for this package content and issue an alert as soon as these packets are detected. This is achieved with the rules shown in Listing 5.7.

```
alert tcp any any -> any any (msg:"Flow␣flood␣detected"; GID:1; sid:10000006;
    rev:001; content:"|70␣69␣6E␣67|";)
alert tcp any any -> any any (msg:"Flow␣flood␣detected"; GID:1; sid:10000007;
    rev:001; content:"|70␣6F␣6E␣67|";)
```

Listing 5.7: Flow flood rule

This method relies solely on packet content and a known signature of the attack. If this is the case, it can generate alerts as soon as a matching packet is detected. In this implementation, this can be as soon as the first packet containing `ping` arrives at the monitoring instance.

A second way of detecting a flow flood with an IDS would be by watching for missing traffic. If a known host has a constant amount of traffic and manipulation is happening inside the flow tables, communication to this host may break down. This difference in packets transmitted can be seen by an IDS with the rule proposed in Listing 5.8. As Suricata is currently missing a feature for matching below instead of above a threshold in its rule set,

Figure 5.11.: SYN Flood with Statshandler

this is not currently implemented.

```
alert tcp $HOME_NET any -> $HOME_NET any (msg:"Flow␣flood␣detected"; GID:1;
    sid:10000008; rev:001; threshold: type both, track by_dst, count !500,
    seconds 10;)
```

Listing 5.8: Flow flood rule

In summary, Suricata is barely able to detect a flow flooding attack. It is missing the capabilities to measure values directly linked to the attack or has to rely on detecting a very specific and already known signature.

Statshandler

Having access to the internal statistics of the SDN will be a key advantage of Statshandler to detect flow flooding as shown in this section. As with every statistics message sent from the manager to monitoring, a list of installed flows and their statistics is provided, the main value used in this implementation is the number of flows installed to a switch. In case of this rather static test setup, the number of active flows is not varying more than by a few. Therefore, a simple limit on how many more flows are active during the last time period is sufficient. In **??**, a flow flooding attack can be seen with alerts generated by the statistic component.

This implementation uses the `OFPAggregateStats` provided by Ryu to receive a compact representation of active flows. Sorted by switch and port, this allows easy detection of maliciously used OVS ports. An example part of such a statistic message received is shown in Listing 5.9

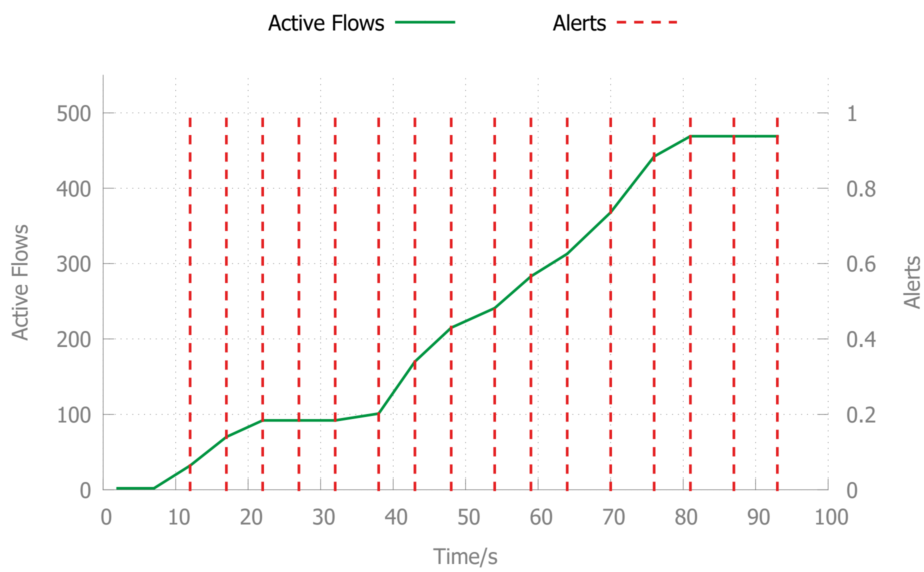

Figure 5.12.: Flow flood with Statshandler

```
"port_aggregate": {
 "0000362b2e9f7f44": {
  "1": [
   {
    "OFPAggregateStatsReply": {
     "flags": 0,
     "type": 2,
     "body": {
      "OFPAggregateStats": {
       "byte_count": 0,
       "flow_count": 0,
       "packet_count": 0
      }
     }
    }
   }
  ]
 }
}
```

Listing 5.9: Port aggregate statistics message

In comparison to Suricata, this is a superior approach. The Statshandler uses fewer resources because there is no package analysis done. Therefore no packages have to be rerouted, the SDN controller only requests statistical information every time interval. There is no dependency to an attack implementation, as soon as a port misbehaves, it is detected. Additionally, the OVS port is specified precisely, therefore even techniques like IP spoofing don't impact the detection. In summary, the uncovering of attacks targeted at SDN specific components requires methods specifically built for this purpose, which Statshandler is.

5.6. Monitoring conclusion

Though the concept of SDN heavily influences every phase of the monitoring process, the single largest contact point of this component with the network is its gathering phase. The way data flows to the component and the type of data that is available to it is specific to a Software Defined Network (SDN) and sufficient to monitor the network. Data present in regular networks like packets per port are also available.

Additionally, SDN provides additional data to give more insight on what paths are used in the network. Both of these are available from a single access point or the single point can be configured to sent information to the monitoring instance. This ease of use is an advantage compared to regular setups.

In its analysis phase, proven techniques can be used to analyze the available standard information and new data can be used to detect anomalies. IDSs already in use still have their place in the network, providing additional information and further analysis of packets. As can be seen with the implemented attacks, the component can analyze and detect more traditional attacks like DDoS as well as new and SDN specific ones like flow flooding.

To output results, the SDN provides a solid foundation and a reliable and configurable way to get the information the next component, the mitigation instance.

6. Mitigation

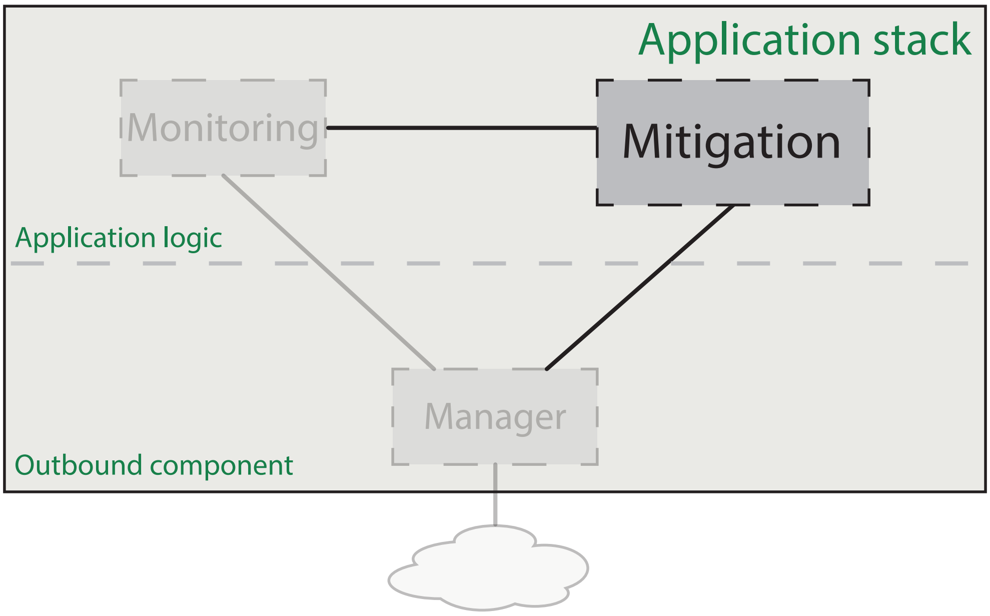

Figure 6.1.: Mitigation in the concept

For mitigation there exist two very different approaches: A proactive mitigation strategy tries to reduce the probability of a successful DDoS attack, by employing techniques like Ingress filtering to preemptively block an attack. On the other hand there is a reactive mitigation strategy that tries to detect an ongoing attack and actively mitigate that particular attack.[DBP05]

Flows are a great way to mitigate attacks. However proactively blocking traffic in a network is for most attacks just not feasible. Especially when considering that a network will most of the time be operating normally. Therefore, as the goal was to explore the capabilities of SDN, the proposed mitigation component uses a reactive approach that can take full advantage of SDN.

As shown in figure 6.1 it fits conceptually between the Monitoring and Management component.

6.1. Mitigation concept

Figure 6.2 shows the process of the mitigation.

The first phase of a successful reactive mitigation, is a trigger that starts the mitigation process and provides the information necessary to handle the ongoing attack. A central database manages all the alerts generated in the network and an Alert Handler, as introduced

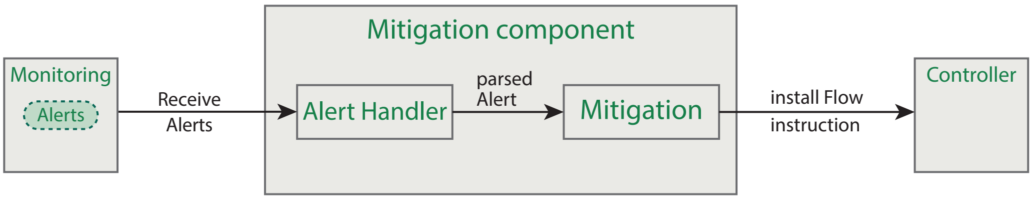

Figure 6.2.: Concept of the mitigation component

in subsection 6.1.2, subscribes to this database to receive the alerts of interest. These alerts serve as the trigger, as well as the source for all the information needed.

Once such an alert is received, the second phase starts. Based on the information provided the appropriate countermeasures have to be chosen from the set of implemented mitigation strategies. In section 6.3 a selection of such mitigation strategies is introduced and implemented.

Finally in the last stage the countermeasures that are implemented via instructions for the creation of flow entries are sent to the controller, who based on these instructions will implement the flow entries on the required devices. In subsection 6.1.4, this is introduced in more detail.

6.1.1. Alerts

First and foremost an alert must be assignable to a problem or an attack in the network. To this end a unique ID, called the Signature ID, is embedded into every alert. Additionally the system that has issued the alert is added. To reliably reconstruct when the alert was raised a timestamp is included as well.

With just this information it is already clearly described what the issue is. However, it is yet unclear where the problem is and who is involved. Based on the Signature ID and source of the alert, there is a set of information the Mitigation module expects to be informed about. This can be information such as the ID of the switch, where an issue has been detected, the IP Address of the victim or the attacker, the port of the switch under attack, or the TCP port that is targeted.

6.1.2. Alert Handler

The Alert Handler is a module that takes care of receiving the alerts for the Mitigation module.

The alerts are divided into multiple channels. As such for example the Intrusion Detection Systems publishes its alerts to a different channel than the system doing statistical monitoring. For each channel the Mitigation module wants to receive the alerts from an instance of this module is created. This instance is responsible for creating a connection to the database and subscribe to the alert channel. Now every alert that is published to the subscribed alert channel is also sent to that instance of the Alert Handler.

After receiving a message it is checked, if it is an alert, and the alert is forwarded to the mitigation module.

6.1.3. Mitigation

In the mitigation module all the alerts coming from the various Alert Handlers are handled.

The Signature ID is extracted from every alert and looked up in the set of available mitigation strategies. Naturally, as this module is specialized in handling attacks, based on a predefined mitigation strategy, only alerts with a known Signature ID can be handled.

If a mitigation strategy is available further actions depend on this strategy. It can be that the information from a single alert is enough to immediately deploy countermeasures, but the information provided from a single alert might also be insufficient for that type of attack. In this case the information contained in the alert is stored together with the timestamp. When the alerts containing the missing information, potentially from different systems and alert channels, are received, the time difference between the arrival of the alerts is calculated. To assure that these alerts are related to the same event only if this falls within the specified range of tolerance the countermeasures are constructed. The countermeasures consist of instructions for the creation or deletion of flows.

The first step to add flows is to create the match fields. This specifies the criteria for which packets to act upon. The values for these fields are mainly supplied by the information gathered from the alerts. Common criteria are related to the target of the attack or the source of the attack, such as the MAC Address or IP Address, or the origin of the attack in the network, such as the port of the switch.

The next step is to define the action of the flow. For example the packets can be dropped immediately or sent to another system for further inspection. Next additional parameters can be specified, such as the soft or hard timeout. A soft timeout deletes the flow automatically after a set time of inactivity, while a hard timeout automatically deletes the flow after the set time, independent of activity. The priority of the flow is another parameter of interest. Usually the highest priority is desired, as this assures that the mitigation flow is the first to match the packet. If it was the case that a flow with a higher priority matches the packet first, the packet would be handled by that flow and would never match the mitigation flow, rendering it completely ineffective.

Finally the switches, the flow should be installed on, are specified. This can either be the ID of a switch or a keyword to specify a set of switches, such as every switch or the switches on the perimeter of the network.

These instructions are then handed over to the Northbound API of the controller.

6.1.4. Controller Northbound API

The Northbound API of the controller provides an API for Network Applications to interact with the controller. As such it also enables the Mitigation module to implement flows on the switches.

To install flow entries on the flow tables the instructions provided have to be translated into OpenFlow commands that can be sent out on the Southbound API of the controller. As for the mitigation of the attacks a powerful API is necessary the translation should be as generic as possible. To simplify the creation of the instructions default values for parameters that are not explicitly set are defined here.

Also the keywords for the set of switches, the flow entries are to be installed on, have to be translated into the corresponding set of actual switches.

6.2. Implementation

This chapter shows how the concept introduced above is realized.

The mitigation component is run as a service on a LXC that is dedicated to mitigation and the modules are implemented fully in python and run as a system service.

6.2.1. Alert Handler

The central database that manages the alerts is Redis that was introduced in subsection 4.4.8.

```
1 [Unit]
2 Description=Alerthandling service
3 After=syslog.target network.target
4
5 [Service]
6 Type=simple
7 ExecStart=/usr/bin/python3 /vagrant/mitigation/AlertHandler.py -r 10.20.0.6 -k switch-stats
8
9 [Install]
10 WantedBy=multi-user.target
```

Listing 6.1: Alert Handler Service Configuration File

For every instance of the alert handler a configuration file, as shown in listing 6.1, is provided. Line 7 shows the command line arguments that are used for the configuration. ”-r 10.20.0.6” specifies the IP Address of the Redis server to connect to and ”-k switch-stats” the name of the alert channel to subscribe to. Currently there are two channels used and as such two instances of this module are started. One for the ”switch-stats” channel that provides the alerts from the statistical monitoring and the other for the ”suricata” channel that provides alerts from the Intrusion Detection System Suricata.

Through the subscribe/publish method used by Redis alerts, sent to one of the channels, are immediately passed to the subscribed Alert Handler. Afterwards they are sent via a RESTful API to the Mitigation module.

The full source code for this module can be found in section B.2.

6.2.2. Mitigation

The mitigation module sets up a HTTP Server that receives the alerts from the Alert Handler.

```
{'host': 'monitoring', 'event_type': 'alert',
'timestamp': '2016-11-26T13:05:25.720206+0100',
'alert': {'signature': 'Syn flood detected', 'severity': 3, 'gid': 1, 'rev': 1, 'signature_id': 10000003},
'proto': 'TCP',
'src_ip': '10.10.10.3', 'src_port': 1234,
'dest_ip': '10.10.10.4', 'dest_port': 21}
```

Listing 6.2: Extract from an alert generated by Suricata

Listing 6.2 shows an alert generated by Suricata. As the alerts are in JSON format they are first translated into a python dictionary.

From that dictionary the Signature ID of the alert is extracted and the appropriate mitigation strategy is found and finally the flow instructions are created.

```
# Create action: Output packet on a port
action['cmd'] = 'OFPActionOutput'
# Paramaters of the action
params['port'] = 2
action['params'] = params
# Add action to the list of actions
actions['action'] = action
# parameters of the OFPMatch object
match['in_port'] = 1
match['eth_dst'] = '00:00:00:00:00:02'
# Put dictionary together
data['cmd'] = 'OFPFC_ADD'
data['actions'] = actions
data['match'] = match
# Specify additional settings
data['priority'] = 1
data['table_id'] = 0
# Send to controller for all switches
send_post(data, 'http://10.20.0.8:8080/switch/all')
```

Listing 6.3: Flow installation on the Northbound API

In listing 6.3 the creation of a flow, such as used in MAC-Learning, via the Northbound API is depicted. This flow matches all packets arriving at port 1 with the destination MAC-Address ”00:00:00:00:00:02” and forwards them to port 2. Also the command to add the flow ”OFPFC_ADD”, the priority ”1” and the number of the flow table ”0” is specified. Finally it is sent to the controller via a RESTful API. To do this the dictionary from above is translated into JSON and sent via a POST to the URL ”http://10.20.0.8:8080/switch/all”, with 10.20.0.8 being the IP Address of the controller. If it shall only be installed on a specific switch the ”all” keyword has to be replaced with the ID of the switch.

The full source code for this module can be found in section B.3.

6.2.3. Controller Northbound API

On the controller the keyword or ID is resolved into the set of switches the instructions are to be installed on. As the controller keeps a dictionary with all the switches in the network it is just a matter of looking up the ID or in the case of the ”all” keyword to just use every switch in the dictionary. As more advanced keywords were not needed for the mitigation strategies presented later on the ”all” keyword is the only one implemented.

```
actions = [parser.OFPActionOutput(2)]
instruction = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,actions)]
match = parser.OFPMatch(in_port=1, eth_dst=00:00:00:00:00:02)
mod = parser.OFPFlowMod(datapath=self.dp, priority=1,table_id=0, match=match,
    instructions=instruction)
self.dp.send_msg(mod)
```

Listing 6.4: Flow installation on the Southbound API

Then these instruction are parsed into OpenFlow commands. For example listing 6.3 is translated to the commands shown in listing 6.4.

Again a packet that arrives on port 1 with the MAC Address ”00:00:00:00:00:02” as destination shall be forwarded to port 2. To achieve this a list of actions is created. In this case the only action is ”OFPActionOutput(2)”: Send the packet out on port 2. Then this list of actions is passed to an instruction function that causes these actions to be applied immediately once the flow is matched.

To configure what the flow matches to a OFPMatch(in_port=1, eth_dst=00:00:00:00:00:02) object is created: Match every packet that arrives on port 1 with the destination MAC Address 00:00:00:00:00:02.

Finally a FlowModification object is created and information, such as the priority of the flow and which table to insert the flow into, is added. Here the default values for MAC Learning flows are chosen.

This object is then sent to the switch, where the flow will be inserted into the flow table.

For the interested reader: Appendix A line 379 shows the method used to parse the dictionary into the OpenFlow commands.

6.3. Mitigation Strategies

In the following the mitigation of three Denial of Service attacks is explored and evaluated. As Denial of Service attacks are one of the main security concerns in any network and are sufficient to demonstrate the mitigation concept introduced, the mitigation of other attacks is left open for future work.

First the concept for the mitigation of the attack is introduced, then the concept is implemented and finally the effectiveness is evaluated.

6.3.1. TCP SYN flood

A technique unique to SDN is the use of flows to apply reactive packet filtering. While a traditional firewall can filter packets as well, it has to be placed on the ingress point of the network. As a result it cannot filter any traffic originating from within that network, as those packets do not pass the firewall. As such it is completely useless against attackers with physical access to the network.[Hu+14] With SDN every switch or router can be used to enforce the packet filtering rules, eliminating any entry points that remain unguarded.

A simple TCP SYN flood originates from only one attacker and can be stopped quite easily. Once the attack has been detected and the IP Address of the attacker has been identified a rule to drop these packets is enough to stop it.[Edd06]

In a SDN environment this is achieved, by setting up a flow with a higher priority than the routing flows, to make sure this flow is the first to get matched and the packet does not get forwarded beforehand. The matching criteria are set to any TCP packet with the source IP Address of the attacker and the destination IP Address of the victim. As action dropping the packet is set.

Implementation of the mitigation strategy

Figure 6.3 shows the flow of events from when a TCP SYN Flood attack is launched against the network until it is mitigated.

Once Monitoring has detected the attack an alert from Suricata, such as the one shown in listing 6.2, is sent to the Mitigation module. This alert contains the information that there is a TCP SYN Flood going on and the IP Address of the attacker and the victim.

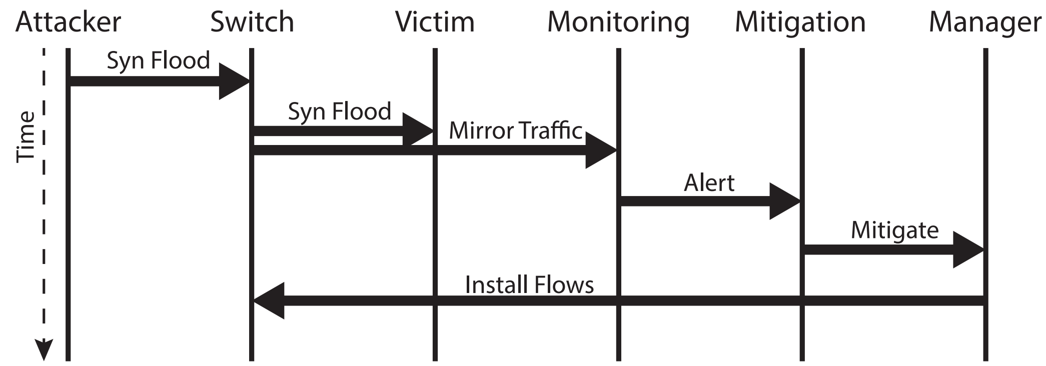

Figure 6.3.: Process of a TCP SYN Flood Mitigation

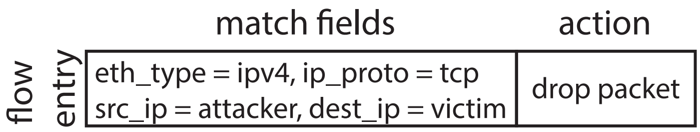

Figure 6.4.: Mitigation flow for a TCP SYN Flood

As this is already enough information to mitigate this attack a flow entry, such as shown in figure 6.4 that stops the attack is assembled immediately and sent to the controller, who propagates it to all the switches in the test setup.

In order to remove the flow again and not keep the flow in the flow table forever, potentially even accumulating too many flow entries at some point, a hard timeout is added. This value specifies the amount of time after which the flow is removed from the switch again automatically. Another reason for not keeping the flow forever is that the IP Address might be reassigned and thus it might end up blocking legitimate traffic. For test purposes a hard timeout of 20 seconds was chosen. But in a real environment a longer timeout is probably more beneficial.

Once the flow has been implemented the first switch of the test setup, the packets arrive at, drops all the packets of the attacker and the SYN Flood packets no longer arrive at the victim.

Evaluation

To evaluate the mitigation strategy a TCP SYN Flood script is started on the attacker LXC with the target LXC as destination. During the attack there is no additional traffic generated in the network.

In order to get data, to compare the performance of the mitigation strategy to, the mitigation module is disabled first. Afterwards the same attack is started once again. But this time the mitigation module is enabled.

Figure 6.5 shows the amount of packets arriving at the Monitoring component during the attack. As the resolution of the packet logging is 3 seconds the graph shows the amount of packets that arrived in the last 3 seconds.

The curve with active mitigation shows clearly that after a short burst of packets arriving the attack is detected and successfully mitigated, as all the packets are dropped.

Every approximately 20 seconds there is small spike, where a few packets can be detected

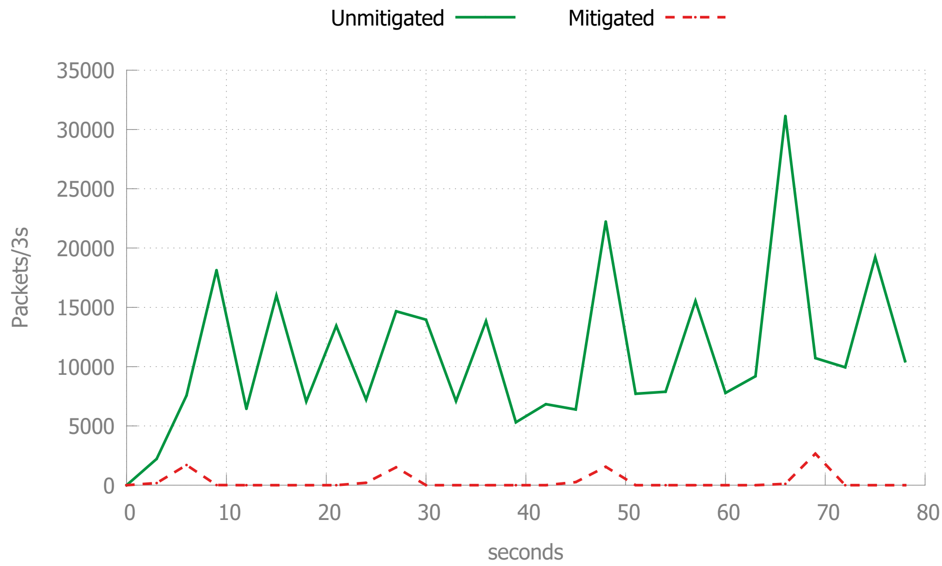

Figure 6.5.: Network load during a TCP SYN flood with and without mitigation

again. These spikes align perfectly with the hard timeout of the mitigation flows. Thus representing the packets arriving during the time the mitigation flow is automatically removed and the attack being detected and mitigated once more, as can be seen clearly in figure 6.6.

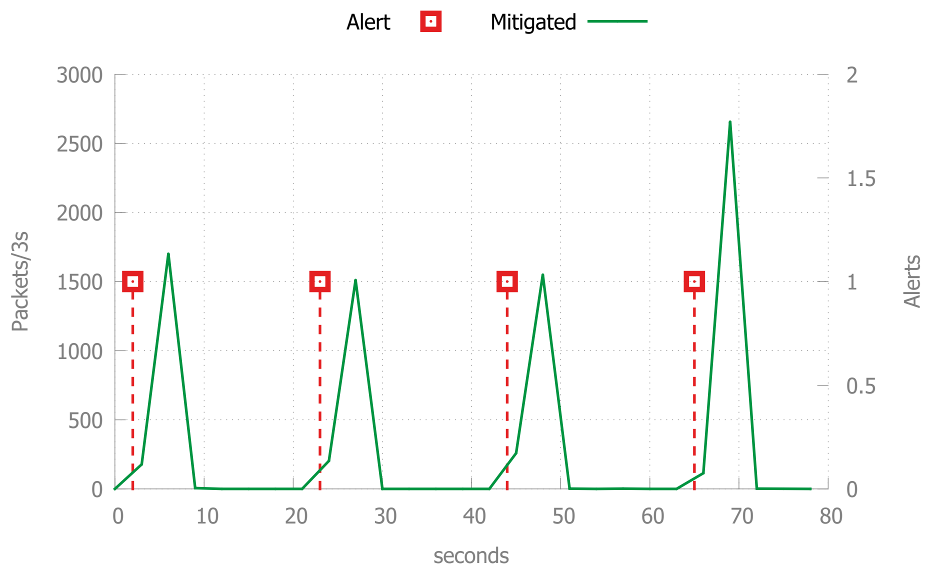

Figure 6.6.: Alerts during mitigated TCP SYN Flood

6.3.2. Distributed TCP SYN Flood

Once the attacker uses a large enough botnet, or especially if he is using IP Spoofing, to forge the source IP Address, the mitigation strategy presented above is no longer capable of defending against the attack.

With spoofed IP Addresses the alerts used previously do no longer trigger, as it relied on many SYN packets coming from a single IP Address. But while the IDS cannot provide information about the source of the attack it can still detect the destination IP Address and the TCP ports under attack.

However, the monitoring component is able to detect the switch and port, the attack is originating from, by monitoring the statistics of the flow tables similar to the technique used in [Cui+16]. This information can be used to drop the traffic at the ingress switch. It is important to install the flow only on the ingress switch, as the flow rules are not as finely granulated, as is the case with a TCP SYN Flood originating from only a single source.

As a result installing such a flow on all devices might block a lot of legitimate traffic to the target.

Implementation of the mitigation strategy

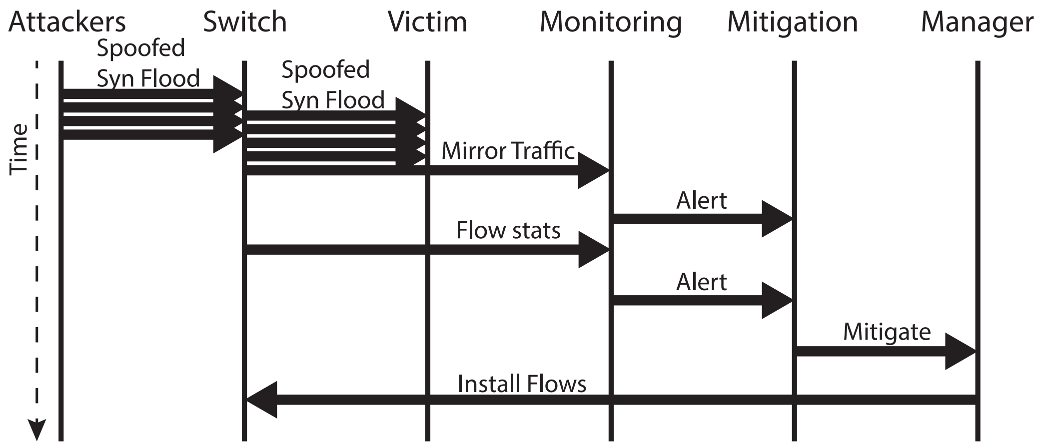

Figure 6.7.: Process of a Distributed TCP SYN Flood Mitigation

In comparison to the simple TCP SYN Flood, shown in figure 6.3, the main difference in the process of mitigating the distributed attack, depicted in figure 6.7, lies in the alerts triggering the mitigation.

The alert, received from Suricata, is pretty similar to the one displayed in listing 6.2. The only real difference is that the Signature ID now signals a Distributed TCP SYN Flood and, while there is still a source IP Address listed in the alert, it is only one of the many IP Addresses used for the attack.

```
{'pps': 0.0, 'event_type': 'alert', 'critical': 500, 'sensor': 'statshandler',
'switch': '0000ee132ef17348', 'alert': {'signature_id': '10000004'},
'port': '4', 'timestamp': '2016-11-30T17:27:44.418396+0100'}
```

Listing 6.5: Alert generated by statistical monitoring

The other part of the information is provided through the alert generated by the statistical monitoring. Such an alert, generated during an attack, is listed in listing 6.5. From this alert the information on which switch, identified by the unique ID "0000ee132ef17348", and on which port the attack (4) is arriving at can be extracted.

Figure 6.8 depicts the workflow for creating the Mitigation flow. Once a corresponding alert arrives the alert is saved. If an alert from Suricata and from the statistical monitoring has arrived within a predefined time frame, for this implementation a time frame of 5 seconds

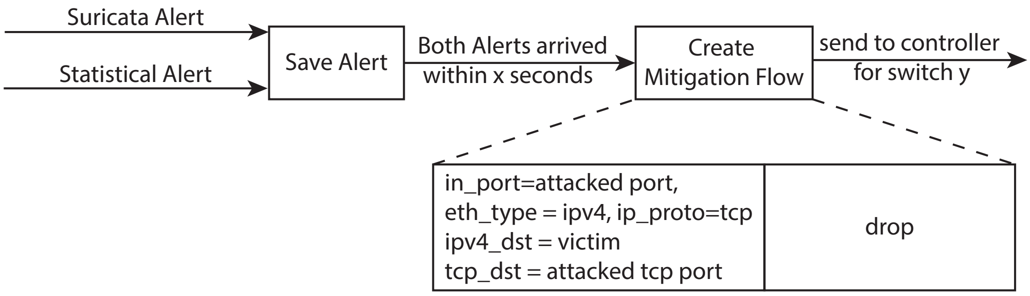

Figure 6.8.: Workflow for the mitigation of a Distributed TCP SYN Flood

was chosen, the mitigation flow is assembled from the information contained in both alerts and sent to the identified switch.

Evaluation

To simulate a Distributed TCP SYN Flood the attack is launched from 2 machines simultaneously. Additionally all the source IP Addresses of the SYN packets are spoofed. While the amount of machines used is not representative for a botnet that might be used in a real attack, given the limited resources available in a virtual machine and by using spoofed IPs, it should nevertheless be representative.

To test if the attack is able to prevent clients from establishing a TCP connections with the victim, a webserver is installed on the LXC of the victim.

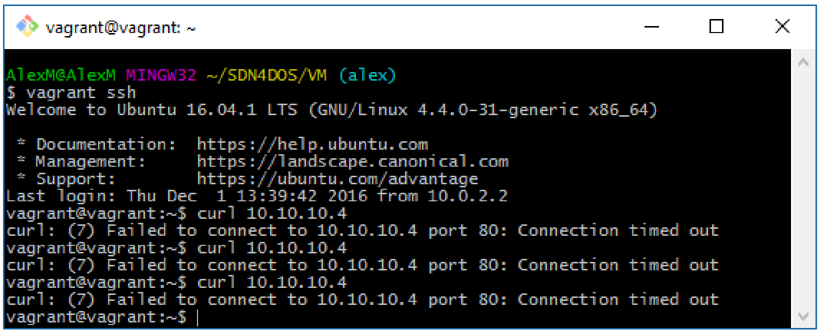

Figure 6.9.: Querying the webserver during an attack

Figure 6.9 shows attempts to query the webserver during the attack. None of the requests manage to successfully establish a connection, proofing that the attack was successful.

Figure 6.10 shows the amount of packets originating from the attack. Without any mitigation it stays at about 10 000 packets per 3 seconds. The dashed line shows the attack with the mitigation strategy discussed above implemented. As the mitigation module has to wait for the alert from the statistical monitoring, which only receives statistical information from the switches every 5 seconds, the mitigation takes a bit longer.

This can be seen better in figure 6.11. Once the attack starts, Suricata is able to detect the target of the attack very quickly and raises an alert. But the alert from the statistical alert still takes some to time to arrive. The hard timeout for the mitigation flow is 20 seconds again, as can be seen by looking at the duration between the 2 peaks, where packets arrive again. As the attack has been executed from two machines that are connected to different

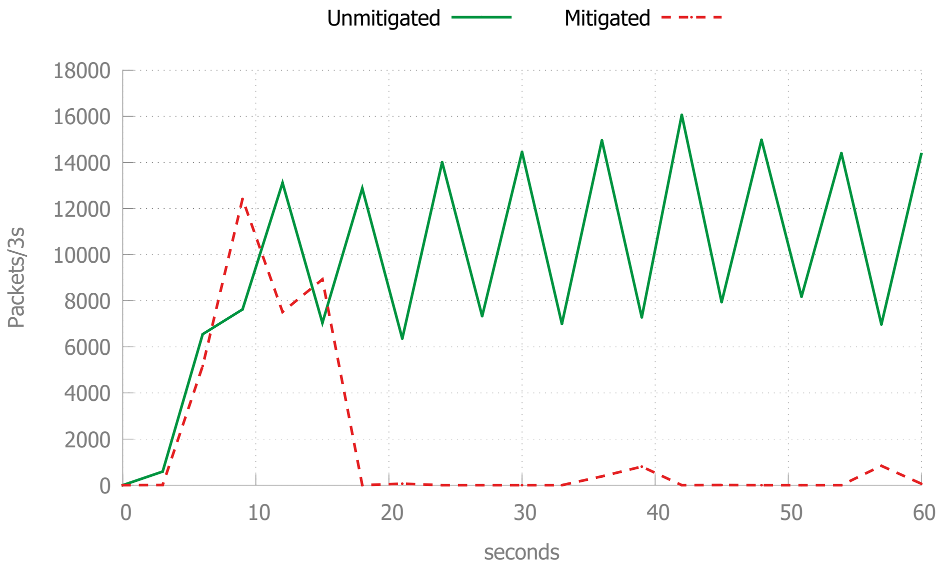

Figure 6.10.: Network load during a distributed TCP SYN flood with and without mitigation

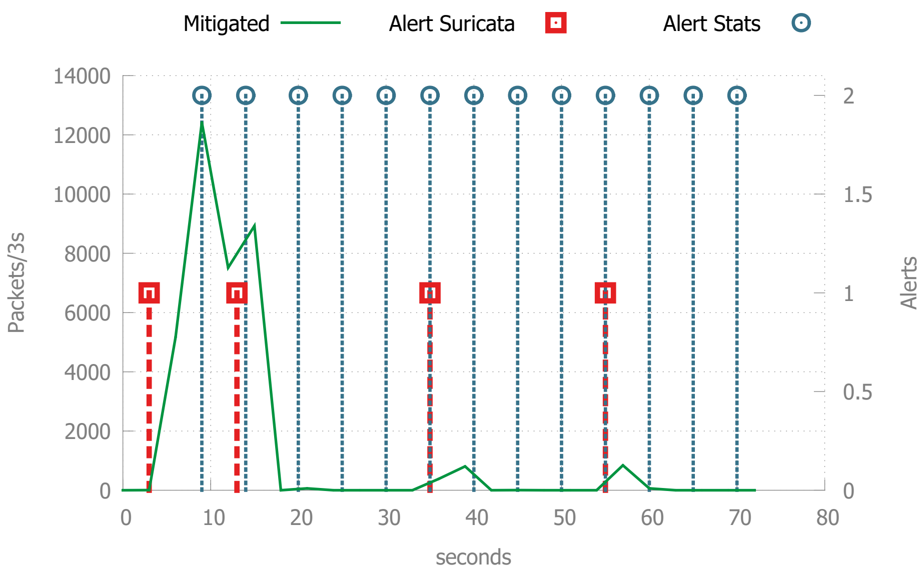

Figure 6.11.: Alerts during mitigated Distributed TCP SYN Flood

ports on the switch.

As the switch still receives packets on the attacked port, monitoring continues to generate alerts even though the packets are dropped. This has the advantage that if the attack is still ongoing, while the flow is deleted through the hard timeout, the alert coming from Suricata can immediately reinstall the flow.

It is to be noted that after the attack has been mitigated the webserver is reachable once again. Thus showing that the mitigation was effective.

6.3.3. Flow flooding

As a Denial of Service attack, this attack shares some inherent similarities with a Distributed TCP SYN Flood. To overflow the flow tables the attacker manipulates packets, usually through or combined with spoofing.

Therefore to mitigate the attack these malicious packets have to be dropped before they can cause a flow rule to be created. As most likely malicious flows have already been created before the mitigation was active these flows have to be removed from the flow table again.

The mitigation component can detect the switch under attack and determine the port the attack is originating from. However, as the attacker does not necessarily target a specific device it is difficult to narrow down the attack. As such before the attack destabilizes the whole network, a first step is to drop all packets arriving on that port. As the rules for the flow creation that are abused may vary from system to system a further analysis of abusable flow creation rules is advisable and the mitigation and detection can be custom tailored to drop only packets that could abuse these vulnerabilities. Thus reducing the amount of legitimate traffic being blocked

After closing the port all flows originating from that port are removed.

Implementation of the mitigation strategy

The detection for this attack is handled by the statistical monitoring. The alert that is received for this kind of attack contains only the information that a Flow flood was detected, the flow table of which switch is filled with malicious flows, and from which port the attack is coming from. Similar to the alert in listing 6.5.

Based on this information the flow to drop the packets is created and sent to the controller. The flow matches every packet that arrives on that port and drops it. As a hard timeout for the flow 20 seconds is chosen again. Thus effectively closing the port for 20s.

Now that the creation of additional superfluous flow entries has been halted the flow entries with in_port="closed port" are to be removed. To prevent the mitigation flow from being deleted a cookie has to be used.

```
data['cookie'] = 0x0000000000000001
data['cookie_mask'] = 0x000000000000000f
```

Listing 6.6: Setting cookies on flows

Listing 6.6 shows how to set the cookie. To remove the other flows instead of the "OFPFC_ADD" command "OFPFC_DELETE" is used and the same matching criteria are used again (in_port="closed port"). But the cookie "0x0000000000000000" is set, while the mask stays the same. This results in all flows without a cookie set and a match containing the port to be removed from the flow table.

Evaluation

First a flow flooding attack was started to take a look at the effect of the attack on the network. Figure 6.12 shows that as the maximum number of 184 flows was reached the response time went up steadily. A look at the CPU load of the controller also showed that as soon as the flow flood was started the load went up to 85%, reaching a steady 100% as the flow table was full. This effect was expected as the controller itself has to handle every

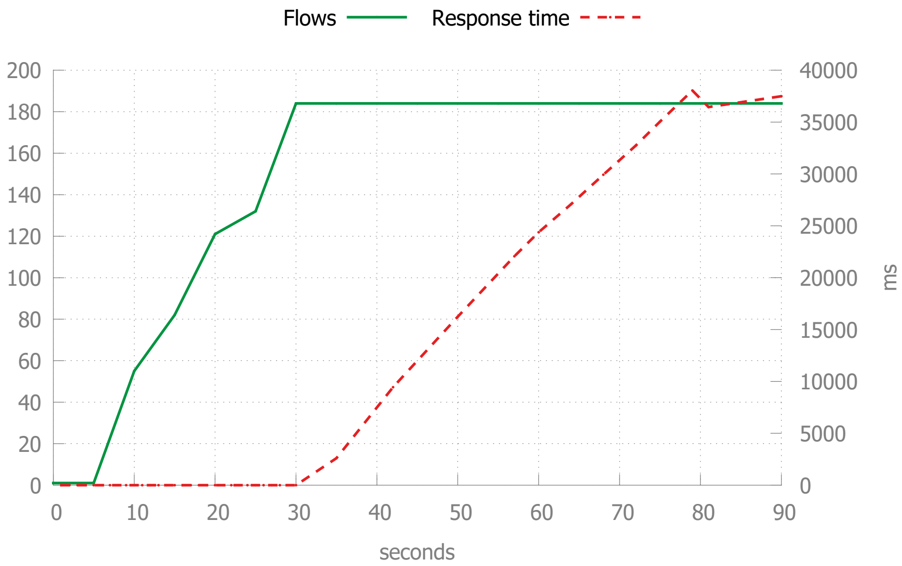

Figure 6.12.: Response time during flow flooding attack

single packet, when no flow rule can be installed. Thus eventually being overwhelmed by the amount of packets that arrive. Even without any load on the network the response time in the network went up from averaging 0.13 ms to 18 ms, showing the overhead introduced through having the controller handle the packets.

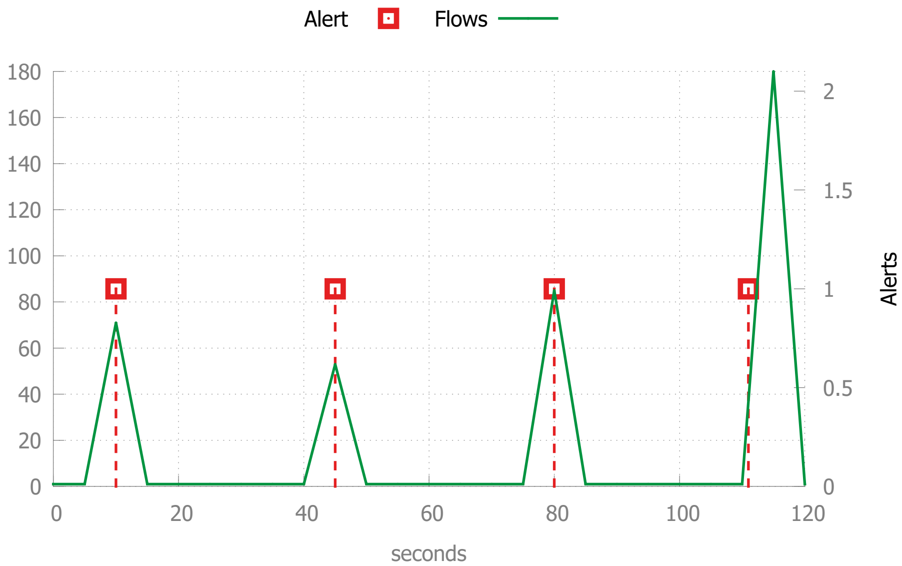

Figure 6.13.: Mitigation of a flow flooding attack

Afterwards the attack was launched again, while the mitigation strategy, presented above, was applied. Figure 6.13 shows that the generation of the flows was stopped before the flow table was even full. Only the last attack has been able to almost fill the flow table. But as the malicious flows are immediately removed again even if the attack manages to flood the table the controller functionality is restored.

6.4. Conclusion

A mitigation module was proposed that can handle various attacks based on alerts. In that context it was shown that flows serve as a powerful and versatile tool for the mitigation of attacks. Through the use of alerts and the Alert Handler, the module can be easily extended to handle alerts from various sources.

To install flows via the Northbound interface a powerful API was implemented that provides nearly the same functionality as if creating the flow instructions directly on the controller, by translating a python dictionary with a syntax closely related to the one used on the Southbound API into the respective OpenFlow commands.

The usability of the module has been shown by providing a mitigation strategy for a TCP SYN Flood, its distributed counterpart and a Flow Flooding attack. The results show that the mitigation module is able to quickly mitigate the attacks.

7. Conclusion and future work

An application stack for detection and mitigation of attacks in SDN was proposed and demonstrated. For the implementation of the application stack a test environment that can produce reproducible results was realized through a virtual machine based on a LXC architecture that is provided through vagrant. The test environment was then used to evaluate the proposed detection and mitigation mechanism.

A way of enhancing the testing environment can be to implement more caching capabilities. As a significant amount of the setup process is dependent on having access to the Internet, caching more downloaded packages can speed up the process and provide a faster to use setup. Additionally, the testing workflow could be enhanced by automating the data exported from the visualization instance. While sufficient graphs are available in Grafana to provide insight into the live status of the system, this could ease the workflow of generating high quality plots for further use. To move more into the direction of a production ready setup, security policies have to be enforced and components tested against real traffic. Parameter tuning has to be done and the network layout adapted to the new environment.

The monitoring instance used exemplifies the combination of using software already developed and originally running in a traditional network, and new one specifically developed for the analysis of SDN related data. It proves that already established ways of monitoring data can be used along with new methods to achieve detection of an attack. Both of these can provide valuable information on SDN specific and nonspecific attacks to be used together in order to defend against them.

Improving upon the monitoring component could be done in the future in two main directions. To enable it to detect more complicated and variable attacks, the analysis phase could be extended and already existing detection enhanced. Doing this can involve implementation of new logic to detect attack vectors or exchanging the stiff limit based implementation for a more flexible and adaptive one.

Though this would introduce more load to the instance, the second option could be to add an additional phase. This phase could be a lightweight component looking for any suspicious behavior and inserting a full monitoring instance into the path of traffic only after such behaviour is detected. This would improve resource usage by eliminating a lot of analysis time now spent on regular packets.

In the mitigation chapter a concept for mitigating attacks through the use of flows was introduced. This was achieved by creating flow rules from the information provided from the alerts raised by the monitoring module that block the attack. This was successfully proven on the example of three attacks.

A TCP SYN Flood attack was mitigated by installing a flow that blocks packets from the attacker on the way to the victim. While it was shown that the proposed mitigation blocks the attack, an attacker could spoof the IP Address of a system in the network to cause the mitigation to block the traffic of that system. To prevent this it might prove useful to implement source verification, such as VAVE [YBX11].

In the next step a Distributed TCP SYN Flood was tackled. Through the use of information provided by the flow statistics the ingress switch was identified and the malicious packets could be stopped, as soon as they entered the network. A known problem is that high traffic can be mistaken for a SYN flood attack. As the controller is written with support for up to OpenFlow version 1.4, matching the SYN flag in a TCP packet was not yet possible. As such supporting OpenFlow version 1.5 and matching on the SYN flag is a task that remains open.

Finally a Flow Flooding attack was mitigated. However more extensive research in regard to flow rules used in a production environment that are susceptible to this attack is to be done. This could also provide further insight on how the mitigation flows can be grained more finely to avoid blocking legitimate clients from accessing the network.

The goal set in chapter 1 to develop an SDN aware application stack has been reached and proven its capabilities while using the test setup, which was also implemented during the process. Further works can use this as a base to build upon.

A. Controller

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

import logging
import struct
import socket
import redis
import time

import json
from webob import Response

from threading import Timer

from ryu.app.wsgi import ControllerBase
from ryu.app.wsgi import WSGIApplication
from ryu.base import app_manager
from ryu.controller import dpset
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.exception import OFPUnknownVersion
from ryu.exception import RyuException
from ryu.lib import dpid as dpid_lib
from ryu.lib import hub
from ryu.lib import mac as mac_lib
from ryu.lib import addrconv
```

```
from ryu.lib.packet import packet
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.ofproto import ether
from ryu.ofproto import ofproto_v1_3
from ryu.ofproto import ofproto_v1_4

UINT32_MAX = 0xffffffff
UINT64_MAX = 0xffffffffffffffff

ARP = arp.arp.__name__

OFP_REPLY_TIMER = 1.0 # sec

SWITCHID_PATTERN = dpid_lib.DPID_PATTERN + r'|all'

COOKIE_DEFAULT_ID = 0

REST_COMMAND_RESULT = 'command_result'
REST_RESULT = 'result'
REST_DETAILS = 'details'
REST_OK = 'success'
REST_NG = 'failure'
REST_ALL = 'all'
REST_SWITCHID = 'switch_id'
REST_COMMAND = 'cmd'
REST_MATCH = 'match'
REST_COOKIE = 'cookie'
REST_COOKIEMASK = 'cookie_mask'
REST_ACTIONS = 'actions'
REST_ACTION = 'action'
REST_PARAMS = 'params'
REST_PRIORITY = 'priority'
REST_TABLEID = 'table_id'
REST_IDLETIMEOUT = 'idle_timeout'
REST_HARDTIMEOUT = 'hard_timeout'

IP_MONITORING = '10.10.10.5'
IP_SWITCHARP = '10.10.10.99'
REDIS_HOST = '10.20.0.6'
REDIS_PORT = 6379

DELAY_CONNECTION = 5.0 # sec

STATS_POLL_RATE = 5 # sec

```

```python

class NotFoundError(RyuException):
  message = 'Switch SW is not connected. : switch_id=%(switch_id)s'


class RestSwitchAPI(app_manager.RyuApp):
  OFP_VERSIONS = [ofproto_v1_4.OFP_VERSION]

  _CONTEXTS = {'dpset': dpset.DPSet,
         'wsgi': WSGIApplication}

  def __init__(self, *args, **kwargs):
    super(RestSwitchAPI, self).__init__(*args, **kwargs)

    SwitchController.set_logger(self.logger)

    wsgi = kwargs['wsgi']
    self.waiters = {}
    self.data = {'waiters': self.waiters}

    mapper = wsgi.mapper
    wsgi.registory['SwitchController'] = self.data
    requirements = {'switch_id': SWITCHID_PATTERN}

    # No vlan data
    path = '/switch/{switch_id}'
    mapper.connect('switch', path, controller=SwitchController,
            requirements=requirements,
            action='get_data',
            conditions=dict(method=['GET']))
    mapper.connect('switch', path, controller=SwitchController,
            requirements=requirements,
            action='set_data',
            conditions=dict(method=['POST']))

    mapper.connect('switch', path, controller=SwitchController,
            requirements=requirements,
            action='delete_data',
            conditions=dict(method=['DELETE']))

  @set_ev_cls(dpset.EventDP, dpset.DPSET_EV_DISPATCHER)
  def datapath_handler(self, ev):
    if ev.enter:
      SwitchController.register_switch(ev.dp, self.waiters)
    else:
      SwitchController.unregister_switch(ev.dp)

```

```
  @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
  def packet_in_handler(self, ev):
    SwitchController.packet_in_handler(ev.msg)

  def _stats_reply_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath

    if dp.id not in self.waiters or msg.xid not in self.waiters[dp.id]:
      return
    event, msgs = self.waiters[dp.id][msg.xid]
    msgs.append(msg.to_jsondict())
    if ofproto_v1_3.OFP_VERSION == dp.ofproto.OFP_VERSION or ofproto_v1_4.OFP_VERSION == dp.ofproto.OFP_VERSION:
      more = dp.ofproto.OFPMPF_REPLY_MORE
    else:
      more = dp.ofproto.OFPSF_REPLY_MORE

    if msg.flags & more:
      return
    del self.waiters[dp.id][msg.xid]
    event.set()

  # for OpenFlow version1.0
  @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
  def stats_reply_handler_v1_0(self, ev):
    self._stats_reply_handler(ev)

  # for OpenFlow version1.2+
  @set_ev_cls(ofp_event.EventOFPStatsReply, MAIN_DISPATCHER)
  def stats_reply_handler_v1_2(self, ev):
    self._stats_reply_handler(ev)

  @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
  def port_stats_reply_handler(self, ev):
    self._stats_reply_handler(ev)

  @set_ev_cls(ofp_event.EventOFPAggregateStatsReply, MAIN_DISPATCHER)
  def aggregate_stats_reply_handler(self, ev):
    self._stats_reply_handler(ev)


def rest_command(func):
  def _rest_command(*args, **kwargs):
    try:
      msg = func(*args, **kwargs)
      return Response(content_type='application/json',
```

```
               body=json.dumps(msg))

  except SyntaxError as e:
    status = 400
    details = e.msg
  except (ValueError, NameError) as e:
    status = 400
    details = e.message

  except NotFoundError as msg:
    status = 404
    details = str(msg)

  msg = {REST_RESULT: REST_NG,
      REST_DETAILS: details}
  return Response(status=status, body=json.dumps(msg))

 return _rest_command


class SwitchController(ControllerBase):
 _SWITCH_LIST = {}
 _LOGGER = None

 def __init__(self, req, link, data, **config):
  super(SwitchController, self).__init__(req, link, data, **config)
  self.waiters = data['waiters']

 @classmethod
 def set_logger(cls, logger):
  cls._LOGGER = logger
  cls._LOGGER.propagate = False
  hdlr = logging.StreamHandler()
  fmt_str = '[RT][%(levelname)s] switch_id=%(sw_id)s: %(message)s'
  hdlr.setFormatter(logging.Formatter(fmt_str))
  cls._LOGGER.addHandler(hdlr)

 @classmethod
 def register_switch(cls, dp, waiters):
  dpid = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
  try:
   switch = Switch(dp, cls._LOGGER, waiters)
  except OFPUnknownVersion as message:
   cls._LOGGER.error(str(message), extra=dpid)
   return
  cls._SWITCH_LIST.setdefault(dp.id, switch)
  cls._LOGGER.info('Join as switch.', extra=dpid)
```

```
        # Try to setup connection to Monitoring after x seconds
        t = Timer(DELAY_CONNECTION, switch.connect_monitoring)
        t.start()

    @classmethod
    def unregister_switch(cls, dp):
        if dp.id in cls._SWITCH_LIST:
            cls._SWITCH_LIST[dp.id].delete()
            del cls._SWITCH_LIST[dp.id]

            dpid = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
            cls._LOGGER.info('Leave switch.', extra=dpid)

    @classmethod
    def packet_in_handler(cls, msg):
        dp_id = msg.datapath.id
        if dp_id in cls._SWITCH_LIST:
            switch = cls._SWITCH_LIST[dp_id]
            switch.packet_in_handler(msg)

    # GET /switch/{switch_id}
    @rest_command
    def get_data(self, req, switch_id, **_kwargs):
        return self._access_switch(switch_id, 'get_data', req)

    # POST /switch/{switch_id}
    @rest_command
    def set_data(self, req, switch_id, **_kwargs):
        return self._access_switch(switch_id, 'set_data', req)

    # DELETE /switch/{switch_id}
    @rest_command
    def delete_data(self, req, switch_id, **_kwargs):
        return "Not implemented"

    def _access_switch(self, switch_id, func, req):
        rest_message = []
        switches = self._get_switch(switch_id)
        try:
            param = req.json if req.body else {}
        except ValueError:
            raise SyntaxError('invalid syntax %s', req.body)
        for switch in switches.values():
            function = getattr(switch, func)
            data = function(param)
            rest_message.append(data)

```

```python
        return rest_message

    def _get_switch(self, switch_id):
        switches = {}

        if switch_id == REST_ALL:
            switches = self._SWITCH_LIST
        else:
            sw_id = dpid_lib.str_to_dpid(switch_id)
            if sw_id in self._SWITCH_LIST:
                switches = {sw_id: self._SWITCH_LIST[sw_id]}

        if switches:
            return switches
        else:
            raise NotFoundError(switch_id=switch_id)


class Switch(dict):
    def __init__(self, dp, logger, waiters):
        super(Switch, self).__init__()
        self.dp = dp
        self.dpid_str = dpid_lib.dpid_to_str(dp.id)
        self.sw_id = {'sw_id': self.dpid_str}
        self.logger = logger

        self.monitoring_port = None

        self.port_data = PortData(dp.ports)

        self.history = {}

        self.mac_to_port = {}

        self.ofctl = OfCtl.factory(dp, logger)

        self.setup_basic_flows()

        self.redis_instance = self.setup_redis()

        self.monitor_thread = hub.spawn(self._monitor, waiters)

    def _monitor(self, waiters):
        """Requesting statistic messages from switches at STATSPOLLRATE

        :param waiters: the switches to request stats from
        """
```

```

        while True:
            stats = self.request_stats(waiters)
            message = {
                'event_type': 'statistics',
                'statistics': stats['statistics'],
            }
            self.redis_instance.publish('switch-stats', json.dumps(message))
            hub.sleep(STATS_POLL_RATE)

    def setup_redis(self, retry=0, max_retries=30):
        try:
            redis_instance = redis.StrictRedis(host=REDIS_HOST, port=REDIS_PORT, db=0)
        except redis.exceptions.ConnectionError as e:
            if retry < max_retries:
                self.logger.warning("Retrying connections to database for the {} time".format(retry + 1),
                                    extra=self.sw_id)
                time.sleep(2)
                redis_instance = self.setup_redis(retry=retry + 1)
            else:
                raise e

        return redis_instance

    def connect_monitoring(self):
        # Send ARP Request to get MAC-Address of Monitoring
        self.send_arp_request(IP_SWITCHARP, IP_MONITORING)
        self.logger.info('Send ARP request (flood)', extra=self.sw_id)

    def delete_flows(self, table_id):
        # Delete all flows in Table
        # Used to delete old flows before monitoring was connected
        ofproto = self.dp.ofproto
        parser = self.dp.ofproto_parser
        match = parser.OFPMatch()
        inst = []
        mod = parser.OFPFlowMod(self.dp, 0, 0, table_id, ofproto.OFPFC_DELETE, 0, 0, 1, ofproto.OFPCML_NO_BUFFER,
                                ofproto.OFPP_ANY, ofproto.OFPG_ANY, 0, 0, match, inst)
        self.dp.send_msg(mod)

    def setup_basic_flows(self):
        ofproto = self.dp.ofproto
        parser = self.dp.ofproto_parser

        # Table Miss Entry
```

```
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                     ofproto.OFPCML_NO_BUFFER)]
    self.ofctl.add_flow(0, 0, match, actions)

    # Make sure our ARP replies are sent to the controller

    match = parser.OFPMatch(arp_tpa=IP_SWITCHARP, eth_type=2054)
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                     ofproto.OFPCML_NO_BUFFER)]
    self.ofctl.add_flow(3, 0, match, actions)

  def delete(self):
    self.logger.info('Stopped switch', extra=self.sw_id)

  def get_data(self, data, waiters):
    parser = self.dp.ofproto_parser
    msg = "Not implemented" #TODO return load statistics
    return {REST_SWITCHID: self.dpid_str,
        REST_COMMAND_RESULT: msg}

  def set_data(self, data):
    parser = self.dp.ofproto_parser
    ofp = self.dp.ofproto
    msgs = []
    try:
      if REST_COMMAND in data:
        action_list = []
        cmd = data[REST_COMMAND]
        command_obj = getattr(ofp, cmd)
        if REST_MATCH in data:
          matches = data[REST_MATCH]
          match = parser.OFPMatch(**matches)
        else:
          match = parser.OFPMatch()
        if REST_ACTIONS in data:
          actions = data[REST_ACTIONS]
          if REST_ACTION in actions:
            for k, v in actions.items():
              if REST_COMMAND in v:
                cmd = v[REST_COMMAND]
                action_obj = getattr(parser, cmd)
              else:
                details = "No action Command found"
                raise ValueError(details)
              if REST_PARAMS in v:
```

```
            params = v[REST_PARAMS]
            action_list.append(action_obj(**params))
          else:
            details = "No Parameters provided for %s" % cmd
            raise ValueError(details)
      else:
        details = "No action found in actions"
        raise ValueError(details)
    if REST_PRIORITY in data:
      priority = data[REST_PRIORITY]
    else:
      # Default value for mitigation flows
      priority = 32768
    if REST_TABLEID in data:
      table_id = data[REST_TABLEID]
    else:
      # Default value for mitigation flows
      table_id = 0
    if REST_COOKIE in data:
      cookie = data[REST_COOKIE]
    else:
      cookie = 0
    if REST_COOKIEMASK in data:
      cookie_mask = data[REST_COOKIEMASK]
    else:
      cookie_mask = 0
    if REST_IDLETIMEOUT in data:
      idle_timeout = data[REST_IDLETIMEOUT]
    else:
      idle_timeout = 0
    if REST_HARDTIMEOUT in data:
      hard_timeout = data[REST_HARDTIMEOUT]
    else:
      hard_timeout = 0
    if cmd == 'OFPFC_DELETE':
      self.ofctl.delete_flow(priority=priority, table_id=table_id, match=match, command=command_obj,
                  cookie=cookie, cookie_mask=cookie_mask)
    else:
      self.ofctl.add_flow(priority=priority, table_id=table_id, match=match, action_list=action_list,
                hard_timeout=hard_timeout, idle_timeout=idle_timeout, command=command_obj,
                cookie=cookie, cookie_mask=cookie_mask)
    self.logger.info(
      "Added mitigation flow: {}{}{}{}{}{}".format(priority, table_id, match, action_list,
```

```
                                  hard_timeout, idle_timeout), extra=self.sw_id)
        details = 'Performed %s' % cmd
      else:
        raise ValueError("No command found")
      msg = {REST_RESULT: REST_OK, REST_DETAILS: details}
    except ValueError as err_msg:
      msg = {REST_RESULT: REST_NG, REST_DETAILS: str(err_msg)}
    msgs.append(msg)
    return {REST_SWITCHID: self.dpid_str,
        REST_COMMAND_RESULT: msgs}

  def request_stats(self, waiters):
    stats = {}
    stat_replies = {}
    stat_replies['port_aggregate'] = {}
    stat_replies['port_aggregate'][self.dpid_str] = {}
    for send_port in self.port_data.values():
      send_port = send_port.port_no
      stat_replies['port_aggregate'][self.dpid_str][send_port] =
    self.ofctl.get_all_flow_aggregate(send_port,
                                                                  waiters)
    # flows = self.ofctl.get_all_flow(waiters)
    # if flows:
      # flows = self.ofctl.get_all_flow(waiters)[0]
    # stat_replies['flow'] = {}
    # stat_replies['flow'][self.dpid_str] = flows
    portstats = {}
    ports = self.ofctl.get_all_port(waiters)
    if ports:
      ports = self.ofctl.get_all_port(waiters)[0]
      for msg in ports['OFPPortStatsReply']['body']:
        msg = msg['OFPPortStats']
        port_no = msg['port_no']
        del msg['port_no']
        # Calculate deltas
        msg['rx_bytes_delta'] = self.get_port_delta(msg, port_no, 'rx_bytes')
        msg['rx_dropped_delta'] = self.get_port_delta(msg, port_no, 'rx_dropped')
        msg['rx_errors_delta'] = self.get_port_delta(msg, port_no, 'rx_errors')
        msg['rx_packets_delta'] = self.get_port_delta(msg, port_no, 'rx_packets')
        msg['tx_bytes_delta'] = self.get_port_delta(msg, port_no, 'tx_bytes')
        msg['tx_dropped_delta'] = self.get_port_delta(msg, port_no, 'tx_dropped')
        msg['tx_errors_delta'] = self.get_port_delta(msg, port_no, 'tx_errors')
        msg['tx_packets_delta'] = self.get_port_delta(msg, port_no, 'tx_packets')
        portstats[port_no] = msg
    stat_replies['port'] = {}
    stat_replies['port'][self.dpid_str] = portstats
    stats['statistics'] = stat_replies
```

```
    return stats

  def get_port_delta(self, msg, port, field):
    new_value = msg[field]
    if port in self.history:
      if field in self.history[port]:
        old_value = self.history[port][field]
        self.history[port][field] = new_value
        return (new_value - old_value) / STATS_POLL_RATE
      else:
        self.history[port][field] = new_value
        return new_value / STATS_POLL_RATE
    else:
      self.history[port] = {}
      self.history[port][field] = new_value
      return new_value / STATS_POLL_RATE

  def packet_in_handler(self, msg):
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)

    if not eth:
      self.logger.warning("Non ethernet Packet received!", extra=self.sw_id)
      return
    arp_p = pkt.get_protocol(arp.arp)
    if arp_p:
      self._packetin_arp(msg, arp_p)
    ofproto = self.dp.ofproto
    parser = self.dp.ofproto_parser
    in_port = msg.match['in_port']

    if eth.ethertype == ether_types.ETH_TYPE_LLDP:
      # ignore lldp packet
      return
    dst = eth.dst
    src = eth.src

    # self.logger.info("Packet in [src:%s] [dst:%s] [in_port:%s]", src, dst, in_port,
    #           extra=self.sw_id)

    self.mac_to_port[src] = in_port

    if dst in self.mac_to_port:
      out_port = self.mac_to_port[dst]
    else:
      out_port = ofproto.OFPP_FLOOD

```

```
    # self.logger.info("Additional [out_port:%s]", out_port,
    #         extra=self.sw_id)

  # Check if Monitoring is connected and packet not already being forwarded to Monitoring
  if self.monitoring_port is not None and out_port != self.monitoring_port and \
          in_port != self.monitoring_port and out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
    if out_port == in_port:
      # Only duplicate to monitoring but do not forward
      actions = [parser.OFPActionGroup(group_id=0)]
    else:
      actions = [parser.OFPActionGroup(group_id=0), parser.OFPActionOutput(out_port)]

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                      actions)]

    mod = parser.OFPFlowMod(datapath=self.dp, priority=1,
                table_id=0, match=match, instructions=inst)
    self.dp.send_msg(mod)
  else:
    if out_port == in_port:
      # Do not forward packets to the in_port (only caused by duplicated packet forwarding?)
      self.logger.info("Illegal rule?: in_port: [%s], mac_dst: [%s] -> out_port: [%s]",
              in_port, out_port, dst, extra=self.sw_id)
      return
    actions = [parser.OFPActionOutput(out_port)]

    if out_port != ofproto.OFPP_FLOOD:
      match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
      self.ofctl.add_flow(1, 0, match, actions)

  data = None
  if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data

  out = parser.OFPPacketOut(datapath=self.dp, buffer_id=msg.buffer_id,
              in_port=in_port, actions=actions, data=data)
  self.dp.send_msg(out)

def _packetin_arp(self, msg, header):

  if header.opcode == arp.ARP_REPLY:
    src_ip = header.src_ip
    dst_ip = header.dst_ip
```

```
      src_mac = header.src_mac
      srcip = ip_addr_ntoa(src_ip)
      dstip = ip_addr_ntoa(dst_ip)
      if dst_ip == IP_SWITCHARP and src_mac in self.mac_to_port:
        out_port = self.mac_to_port[src_mac]
        self.monitoring_port = out_port
        # Delete old flows in table 0
        self.delete_flows(0)
        # Setup basic flows again
        self.setup_basic_flows()
        log_msg = 'Receive ARP request from [%s] to switch port [%s].' \
             ' Setting up duplicate packet flow to port [%s]'
        self.logger.info(log_msg, srcip, dstip, out_port, extra=self.sw_id)
        # Setup Group 0 for packet duplication to Monitoring port

        parser = self.dp.ofproto_parser
        ofproto = self.dp.ofproto

        actions = [parser.OFPActionOutput(port=out_port)]

        buckets = [parser.OFPBucket(actions=actions)]

        req = parser.OFPGroupMod(datapath=self.dp, command=ofproto.OFPGC_ADD,
                     type_=ofproto.OFPGT_ALL, group_id=0, buckets=buckets)

        self.dp.send_msg(req)

  def send_arp_request(self, src_ip, dst_ip, in_port=None):

    # Send ARP request from all ports.
    for send_port in self.port_data.values():
      if in_port is None or in_port != send_port.port_no:
        src_mac = send_port.mac
        dst_mac = mac_lib.BROADCAST_STR
        arp_target_mac = mac_lib.DONTCARE_STR
        inport = self.ofctl.dp.ofproto.OFPP_CONTROLLER
        output = send_port.port_no
        self.ofctl.send_arp(arp.ARP_REQUEST, src_mac, dst_mac, src_ip, dst_ip,
                  arp_target_mac, inport, output)


class PortData(dict):
  def __init__(self, ports):
    super(PortData, self).__init__()
    for port in ports.values():
      data = Port(port.port_no, port.hw_addr)
      self[port.port_no] = data
```

```python


class Port(object):
  def __init__(self, port_no, hw_addr):
    super(Port, self).__init__()
    self.port_no = port_no
    self.mac = hw_addr


class OfCtl(object):
  _OF_VERSIONS = {}

  @staticmethod
  def register_of_version(version):
    def _register_of_version(cls):
      OfCtl._OF_VERSIONS.setdefault(version, cls)
      return cls

    return _register_of_version

  @staticmethod
  def factory(dp, logger):
    of_version = dp.ofproto.OFP_VERSION
    if of_version in OfCtl._OF_VERSIONS:
      ofctl = OfCtl._OF_VERSIONS[of_version](dp, logger)
    else:
      raise OFPUnknownVersion(version=of_version)

    return ofctl

  def __init__(self, dp, logger):
    super(OfCtl, self).__init__()
    self.dp = dp
    self.sw_id = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
    self.logger = logger

  def add_flow(self, priority, table_id, match, action_list=None, hard_timeout=0,
    idle_timeout=0, command=0, cookie=0,
         cookie_mask=0):
    # Abstract method
    raise NotImplementedError()

  def send_arp(self, arp_opcode, src_mac, dst_mac,
         src_ip, dst_ip, arp_target_mac, in_port, output):
    # Generate ARP packet

    ether_proto = ether.ETH_TYPE_ARP
```

```
    hwtype = 1
    arp_proto = ether.ETH_TYPE_IP
    hlen = 6
    plen = 4

    pkt = packet.Packet()
    e = ethernet.ethernet(dst_mac, src_mac, ether_proto)
    a = arp.arp(hwtype, arp_proto, hlen, plen, arp_opcode,
          src_mac, src_ip, arp_target_mac, dst_ip)
    pkt.add_protocol(e)
    pkt.add_protocol(a)
    pkt.serialize()

    # Send packet out
    self.send_packet_out(in_port, output, pkt.data, data_str=str(pkt))

  def send_packet_out(self, in_port, output, data, data_str=None):
    actions = [self.dp.ofproto_parser.OFPActionOutput(output, 0)]
    self.dp.send_packet_out(buffer_id=UINT32_MAX, in_port=in_port,
                actions=actions, data=data)
    # TODO: ?Packet library convert to string
    # if data_str is None:
    #   data_str = str(packet.Packet(data))
    # self.logger.debug('Packet out = %s', data_str, extra=self.sw_id)

  def send_stats_request(self, stats, waiters):
    self.dp.set_xid(stats)
    waiters_per_dp = waiters.setdefault(self.dp.id, {})
    event = hub.Event()
    msgs = []
    waiters_per_dp[stats.xid] = (event, msgs)
    self.dp.send_msg(stats)

    try:
      event.wait(timeout=OFP_REPLY_TIMER)
    except hub.Timeout:
      del waiters_per_dp[stats.xid]

    return msgs


@OfCtl.register_of_version(ofproto_v1_4.OFP_VERSION)
class OfCtl_v1_4(OfCtl):
  def __init__(self, dp, logger):
    super(OfCtl_v1_4, self).__init__(dp, logger)

```

```
def add_flow(self, priority, table_id, match, action_list=None, hard_timeout=0,
  idle_timeout=0, command=0, cookie=0,
      cookie_mask=0):
  datapath = self.dp
  ofproto = datapath.ofproto
  parser = datapath.ofproto_parser

  if command == 0:
    command = ofproto.OFPFC_ADD

  inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                      action_list)]

  mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
              table_id=table_id, match=match,
              instructions=inst, hard_timeout=hard_timeout,
              idle_timeout=idle_timeout, command=command, cookie=cookie,
  cookie_mask=cookie_mask)
  datapath.send_msg(mod)

def delete_flow(self, priority, table_id, match, command, cookie, cookie_mask):
  ofp = self.dp.ofproto
  ofp_parser = self.dp.ofproto_parser

  inst = []

  flow_mod = ofp_parser.OFPFlowMod(self.dp, cookie, cookie_mask, table_id, command,
                    0, 0, priority, ofp.OFPCML_NO_BUFFER, ofp.OFPP_ANY,
                    ofp.OFPG_ANY, 0, 0, match, inst)
  self.dp.send_msg(flow_mod)
  self.logger.info('Delete flow [cookie=0x%x]', cookie, extra=self.sw_id)

def get_all_flow(self, waiters):
  ofp = self.dp.ofproto
  ofp_parser = self.dp.ofproto_parser

  match = ofp_parser.OFPMatch()
  stats = ofp_parser.OFPFlowStatsRequest(self.dp, 0, 0, ofp.OFPP_ANY,
                        ofp.OFPG_ANY, 0, 0, match)
  return self.send_stats_request(stats, waiters)

def get_all_port(self, waiters):
  ofp = self.dp.ofproto
  ofp_parser = self.dp.ofproto_parser

  stats = ofp_parser.OFPPortStatsRequest(self.dp, 0, ofp.OFPP_ANY)
  return self.send_stats_request(stats, waiters)
```

```
  def get_all_flow_aggregate(self, port, waiters):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser

    match = ofp_parser.OFPMatch(in_port=port)

    stats = ofp_parser.OFPAggregateStatsRequest(self.dp, 0, ofp.OFPTT_ALL,
    ofp.OFPP_ANY, ofp.OFPG_ANY, 0, 0, match)
    return self.send_stats_request(stats, waiters)


def ipv4_text_to_int(ip_text):
  if ip_text == 0:
    return ip_text
  assert isinstance(ip_text, str)
  return struct.unpack('!I', addrconv.ipv4.text_to_bin(ip_text))[0]


def ip_addr_ntoa(ip):
  return socket.inet_ntoa(addrconv.ipv4.text_to_bin(ip))


def mask_ntob(mask, err_msg=None):
  try:
    return (UINT32_MAX << (32 - mask)) & UINT32_MAX
  except ValueError:
    msg = 'illegal netmask'
    if err_msg is not None:
      msg = '%s %s' % (err_msg, msg)
    raise ValueError(msg)
```

B. Mitigation component

B.1. Documentation

Alert Handler Documentation

Release 1.0.0rc0

Alex Marczinek

Dec 11, 2016

CONTENTS:

CHAPTER

ONE

USAGE DOCUMENTATION

This module can be used in two ways. First, it can be run in as a standalone piece of software, described in *standalone*. Further usage can happen through installing the script as a system service, more in *system_service*.

This module connects to a Redis Database. As Redis uses multiple channels, there can be multiple instances of this module be runninc concurrently, listening to different channels. Messages received from the Redis channel are then parsed and forwarded to the Mitigation module.

1.1 Requirements

Required to run this script is at least Python 3.4 and the package 'redis' ('pip install redis').

1.2 Standalone Usage

The script can be run in standalone mode using the following options:

- -l : The loglevel for the application
- -r : The IP of the Redis instance to connect to
- -p : The port of the Redis instance
- -k : The key/channel to subscribe to
- -d : The database in Redis to subscribe to, usually 0

1.3 System Service

Usage as a system service can happen, a service file for usage with systemd is provided in ./miti-ids.service and ./miti-stats.service. Copying this file to /etc/systemd/system/ enables systemd to start the script. After a 'systemctl daemon-reload' it can be started with 'systemctl start <service-name>' and enabled at boot with 'systemctl enable <service-name>'.

The usage in the .service file is the same as described in *standalone*.

CHAPTER

TWO

WELCOME TO MITIGATION'S DOCUMENTATION!

CHAPTER

THREE

USAGE DOCUMENTATION

This module runs as a system service, as described in *system_service2*.

It creates an HTTP server for a RESTFul API with the purpose of receiving the alerts from the AlertHandler

Based on these alerts a Mitigation strategy is chosen and implemented via the Northbound API of the controller.

3.1 System Service

Usage as a system service can happen, a service file for usage with systemd is provided in ./miti-mit.service. Copying this file to /etc/systemd/system/ enables systemd to start the script. After a 'systemctl daemon-reload' it can be started with 'systemctl start miti-mit' and enabled at boot with 'systemctl enable miti-mit'.

CHAPTER

FOUR

DOCUMENTED MODULES

In the following sections, the modules contained within the AlertHandler and Mitigation project will be documented.

4.1 AlertHandler

`AlertHandler.`**`setup_redis`**(*ip*, *port*, *key*, *retry=0*, *max_retries=30*)

This will connect to a Redis instance and return the connection object.

Establishing a stable connection with a Redis instance is the goal. Therefore, the method will retry at most max_retries times.

Parameters

- **`ip`** – The IP of the Redis server to connect to.
- **`port`** – The port on which Redis is running.
- **`key`** – The redis channel/key to subscribe the a PubSub object to.
- **`max_retries`** – The maximum number of retries after which connection attempts will fail.
- **`retry`** – The current number of retries executed.

Returns A set of (redis_instance, pubsub)

`AlertHandler.`**`parseargs`**()

Parsing arguments passed

Using the built in argparse module, passed arguments are parsed. Simply calling this method is sufficient, the results will be returned.

Returns A namespace object containing all parsed arguments

`AlertHandler.`**`parse_message`**(*message*)

This will parse the messages received from Redis.

Parses the message received from Redis and sends alert messages to the mitigation module. The alerts are sent to http://127.0.0.1:9000/store.json

Parameters **`message`** – The message received from Redis.

`AlertHandler.`**`main`**()

The main method.

First, arguments passed to the system are parsed. Then, logging is set up. The connection to redis is established by calling setup_redis() and the main event loop started.

Returns The system exit code

4.2 Mitigation

`Mitigation.main()`

The main method.

Logging is setup and the HTTP server is started.

Returns The system exit code

class `Mitigation.MyServer`(*request*, *client_address*, *server*)

The HTTP Server receiving the alerts from Alert Handler

`do_POST()`

Handles the POST messages and passes them to mitigation

`mitigate`(*parsed_alert: dict*)

Selects a mitigation strategy based on the Signature ID from the alert.

Depending on the strategy a flow instruction is created, that is then sent to the controller via a POST to 10.20.0.8:8080

Parameters **`parsed_alert`** – The alert from the Alert Handler as a python dictionary

Returns True if the attack was mitigated False if it could not mitigate the attack

PYTHON MODULE INDEX

B.2. Source Code Alert Handler

```python
import argparse
import json
import logging
import sys
import time

import redis

from src.tools import setup_logging, send_post


def parseargs():
    """Parsing arguments passed

    Using the built in argparse module, passed arguments are parsed. Simply calling this method is sufficient,
    the results will be returned.

    :return: A namespace object containing all parsed arguments
    """

    loglevel = {
        'debug': logging.DEBUG,
        'info': logging.INFO,
        'warning': logging.WARNING,
        'error': logging.ERROR,
        'critical': logging.CRITICAL,
    }

    parser = argparse.ArgumentParser()
    parser.add_argument('-l', '--loglevel', help='Loglevels to output: debug, info, warning, error, critical',
                        default='info', choices=loglevel.keys())
    parser.add_argument('-r', '--redis', help='Redis instance to query for alerts', required=True)
    parser.add_argument('-k', '--key', help='Key to subscribe to in redis', default='suricata')
    parser.add_argument('-p', '--port', help='Port to bind to, default is 6379', default=6379, type=int)
    parser.add_argument('-d', '--database', help='Database number to subscribe to', default=0, type=int)
    args = parser.parse_args()

    # Validating arguments
```

```
    args.loglevel = loglevel[args.loglevel]
    return args


def setup_redis(ip, port, key, retry=0, max_retries=30):
    """This will connect to a Redis instance and return the connection object.

    Establishing a stable connection with a Redis instance is the goal. Therefore, the method will retry at most
    max_retries times.

    :param ip: The IP of the Redis server to connect to.
    :param port: The port on which Redis is running.
    :param key: The redis channel/key to subscribe the a PubSub object to.
    :param max_retries: The maximum number of retries after which connection attempts will fail.
    :param retry: The current number of retries executed.
    :return: A set of (redis_instance, pubsub)
    """

    try:
        redInstance = redis.StrictRedis(host=ip, port=port, db=0)
        pubsub = redInstance.pubsub()
        pubsub.subscribe(key)
    except redis.exceptions.ConnectionError as e:
        if retry < max_retries:
            logging.warning("Retrying connections to database for the {} time".format(retry + 1))
            time.sleep(2)
            pubsub = setup_redis(ip, port, key, retry=retry + 1)
        else:
            raise e

    return pubsub


def parse_message(message):
    """This will parse the messages received from Redis.

    Parses the message received from Redis and sends alert messages to the mitigation module.
    The alerts are sent to http://127.0.0.1:9000/store.json

    :param message: The message received from Redis.
    """
    data = message['data']
    if type(data) == int:
```

```
    return

  received_json = json.loads(data.decode())
  if received_json['event_type'] == 'alert':
    logging.info("Received alert :\n{}".format(received_json))
    send_post(received_json, 'http://127.0.0.1:9000/store.json')


def main():
  """The main method.

  First, arguments passed to the system are parsed. Then, logging is set up. The
    connection to redis is established
  by calling setup_redis() and the main event loop started.

  :return: The system exit code
  """
  args = parseargs()
  setup_logging(args.loglevel)

  pubsub = setup_redis(args.redis, args.port, args.key)
  try:
    while True:
      message = pubsub.get_message()
      if message is None:
        time.sleep(0.1)
      else:
        parse_message(message)
  except KeyboardInterrupt:
    logging.info("Crtl+C Pressed. Shutting down.")
    pubsub.close()
    return 0


if __name__ == "__main__":
  sys.exit(main())
```

B.3. Source Code Mitigation

```
#!/usr/bin/env python
import datetime
import sys
import json
import logging
import signal
```

```python

from http.server import BaseHTTPRequestHandler, HTTPServer

from src.tools import setup_logging, send_post

ETH_TYPE_IPv4 = 2048

IP_PROTO_ICMP = 1
IP_PROTO_TCP = 6

hostName = "localhost"
hostPort = 9000


class MyServer(BaseHTTPRequestHandler):
  """The HTTP Server receiving the alerts from Alert Handler"""
  suricata_alert = {}
  stats_alert = {}

  def do_POST(self):
    """Handles the POST messages and passes them to mitigation"""
    if self.path == '/store.json':
      length = self.headers['content-length']

      data = self.rfile.read(int(length))
      data = data.decode()
      print("Got message: {}".format(data))
      received_json = json.loads(data)

      self.send_response(200, 'OK')
      self.end_headers()

      self.mitigate(received_json)

  def mitigate(self, parsed_alert: dict):
    """Selects a mitigation strategy based on the Signature ID from the alert.

    Depending on the strategy a flow instruction is created, that is then sent to the controller via
    a POST to 10.20.0.8:8080

    :param parsed_alert: The alert from the Alert Handler as a python dictionary
    :return: True if the attack was mitigated False if it could not mitigate the attack
    """

    assert type(parsed_alert) == dict
    alert = parsed_alert['alert']
```

```
    sid = str(alert.get('signature_id', "0"))
    try:
      if sid == "10000001" or sid == "10000002":
        # ICMP
        return "ICMP: do nothing"
      elif sid == "10000003":
        logging.info('TCP SYN Flood Attack detected')
        # Parameters:
        hard_timeout = 20
        # SYN Flood
        src = parsed_alert["src_ip"]
        dst = parsed_alert["dest_ip"]
        match = dict()
        match['ipv4_src'] = src
        match['eth_type'] = ETH_TYPE_IPv4
        match['ipv4_dst'] = dst
        match['ip_proto'] = IP_PROTO_TCP
        data = dict()
        data['cmd'] = 'OFPFC_ADD'
        data['hard_timeout'] = hard_timeout
        data['match'] = match
        post_result = send_post(data, 'http://10.20.0.8:8080/switch/all')
        logging.info(
          'Handled SYN Flood (Target: {dst}, Attacker: {src}): Drop packets for
    {timeout}s. [{post}]'.format(
            dst=dst, src=src, timeout=hard_timeout, post=post_result))
        return True
      elif sid == "10000004":
        if 'host' in parsed_alert and parsed_alert['host'] == "monitoring":
          # Alert from Suricata
          self.suricata_alert['timestamp'] = parsed_alert['timestamp']
          self.suricata_alert['dest_ip'] = parsed_alert['dest_ip']
          self.suricata_alert['dest_port'] = parsed_alert['dest_port']
        elif 'sensor' in parsed_alert and parsed_alert['sensor'] == "statshandler":
          # Alert from Statistical Monitoring
          self.stats_alert['timestamp'] = parsed_alert['timestamp']
          self.stats_alert['switch'] = parsed_alert['switch']
          self.stats_alert['port'] = parsed_alert['port']
        else:
          logging.error('Unknown Alert Source')
          return False

        if 'timestamp' in self.suricata_alert and 'timestamp' in self.stats_alert:
          logging.info('Found possible DDoS Syn Attack')
          time_suricata = datetime.datetime.strptime(self.suricata_alert['timestamp'],
                                 "%Y-%m-%dT%H:%M:%S.%f%z")
```

```
        time_stats = datetime.datetime.strptime(self.stats_alert['timestamp'], "%Y-%m-%dT%H:%M:%S.%f%z")
        time_delta = abs((time_suricata - time_stats).total_seconds())
        logging.debug("Having a timedelta of {} seconds".format(time_delta))
        if time_delta < 5:
          # Parameters:
          hard_timeout = 20
          # SYN Flood
          switch = self.stats_alert['switch']
          port = self.stats_alert['port']
          dest_ip = self.suricata_alert['dest_ip']
          dest_port = self.suricata_alert['dest_port']
          match = dict()
          match['tcp_dst'] = int(dest_port)
          match['eth_type'] = ETH_TYPE_IPv4
          match['ipv4_dst'] = dest_ip
          match['ip_proto'] = IP_PROTO_TCP
          match['in_port'] = int(port)
          data = dict()
          data['cmd'] = 'OFPFC_ADD'
          data['hard_timeout'] = hard_timeout
          data['match'] = match
          post_result = send_post(data, 'http://10.20.0.8:8080/switch/' + switch)
          logging.info('Handled Distributed SYN Flood (Target: {dst}, Port: {dst_port}):' \
                 ' Drop packets on port {port} on switch{switch} for {timeout}s. [{post}]' \
                 .format(dst=dest_ip, dst_port=dest_port, port=port, switch=switch,
                     timeout=hard_timeout, post=post_result))
          return True
      # DDoS Syn Flood
      return True
    elif sid == "10000005":
      # Flow flood mitigation
      logging.info('Flow Flood Attack detected')
      # Parameters:
      hard_timeout = 20
      # SYN Flood
      switch = parsed_alert['switch_id']
      port = parsed_alert['bad_port']
      match = dict()
      match['in_port'] = int(port)
      data = dict()
      data['cmd'] = 'OFPFC_ADD'
      data['hard_timeout'] = hard_timeout
      data['match'] = match
      data['cookie'] = 0x0000000000000001
```

```python
        data['cookie_mask'] = 0x000000000000000f
        post_result = send_post(data, 'http://10.20.0.8:8080/switch/' + switch)
        # Delete malicious flow entries
        data['cmd'] = 'OFPFC_DELETE'
        data['cookie'] = 0x0000000000000000
        data['cookie_mask'] = 0x000000000000000f
        post_result2 = send_post(data, 'http://10.20.0.8:8080/switch/' + switch)
        logging.info('Handled Flow Flood:' \
               ' Drop packets {match} on port {port} on switch {switch} for {timeout}s. [{post}]' \
               .format(match=match, port=port, switch=switch,
                   timeout=hard_timeout, post=post_result))
        logging.info('Handled Flow Flood:' \
               ' Delete Flows on port {port} on switch {switch} for {timeout}s. [{post}]' \
               .format(port=port, switch=switch,
                   timeout=hard_timeout, post=post_result2))
        return True
      elif sid == "20000001":
        # Test for statistical monitoring module
        logging.info("Statistical Test: do nothing")
        return False
      else:
        logging.error("SignatureID: {} is not handled".format(sid))
        return False
    except LookupError as e:
      return e.args


def main():
  """The main method.

  Logging is setup and the HTTP server is started.

  :return: The system exit code
  """
  signal.signal(signal.SIGTERM, signal_term_handler)
  setup_logging(logging.INFO)
  myServer = HTTPServer((hostName, hostPort), MyServer)
  logging.info('Server starting bound to {}:{}'.format(hostName, hostPort))
  try:
    myServer.serve_forever()
  except KeyboardInterrupt:
    shutdown()
    return 0
```

```
def shutdown():
  myServer.server_close()
  logging.info('Server shut down')
  sys.exit(0)


def signal_term_handler(signal, frame):
  shutdown()


if __name__ == '__main__':
  sys.exit(main())
```

C. StatsHandler Documentation

Statshandler Documentation

Release 1.0.0rc0

Christoph Girstenbrei

Dec 11, 2016

CONTENTS:

CHAPTER

ONE

USAGE DOCUMENTATION

This module can be used in two ways. First, it can be run in as a standalone piece of software, described in *standalone*. Further usage can happen through installing the script as a system service, more in *system_service*.

1.1 Requirements

Required to run this script is at least Python 3.4 and the package 'redis' ('pip install redis').

1.2 Standalone Usage

The script can be run in standalone mode using the following options:

- -r : The IP of the Redis instance to connect to
- -p : The port of the Redis instance
- -k : The key/channel to subscribe to
- -d : The database in Redis to subscribe to, usually 0
- -a : The critical value (int) when to alert if a port surpasses this in transmitting packages
- -e : Ports not to alert on, usefull for backbone ports transmitting lots of data
- -s : Switch, if statistics should be sent to Redis or not

If the statistics data is published to the Redis channel 'switch-stats', running the script can be as ease as running: ./StatsHandler.py -r <IP>

1.3 System Service

Usage as a system service can happen, a service file for usage with systemd is provided in ./statshandler.service. Copying this file to /etc/systemd/system/statshandler.service enables systemd to start the script. After a 'systemctl daemon-reload' it can be started with 'systemctl start statshandler' and enabled at boot with 'systemctl enable statshandler'.

The usage in the .service file is the same as described in *standalone*.

CHAPTER

TWO

DEVELOPMENT GUIDE

This software is written with extensibility in mind. It tries to follow the Single Responsibility Principle (Robert C. Martin) and making it easy, to develop e.g. a new analyzer for a piece of statistics.

After finishing the setup process, there are three main phases to Statshandler. Receiving and preprocessing messages into available in-memory data. After that, the analysis phase takes place. At last, alerts generated are published. There is an optional forth phase, in which statistics are published to Redis, too.

2.1 New Analyzer

The main extension capability is in the analysis phase, providing a new analyzer to be able to find new attacks.

To develop a new analyzer, it can be added to the src.Analyzer module. It must inherit from src.Analyzer.BaseAnalyzer and build on that. There are three necessary steps in the subclass: - set an alerting function while calling the __init__ method of the superclass - override the analyze method with a custom one - Adding the class to Statshandler.initiate:to_initiate

Setting the alerting function can be done like this: super(MyAnalyzer, self).__init__(alerter=self._alert_limiter, limit=limit) There are three different alering functions implemented at the moment, each requiring their own set of arguments. These are specified in their method documentation.

All analyzers are initiated at startup. Therefore, the new analyzer has to be added to the list Statshandler.initiate:to_initiate with the following Syntax: [(MyAnalyzer, (argument1, ...)), ...]. Then, a analyzer will be instantiated during startup with the arguments provided in the argument set.

All analyzing logic has to start in the analyze method, as it is called automatically by the rest of the program while running in the mainloop.

CHAPTER

THREE

DOCUMENTED MODULES

In the following sections, the modules contained within the Statshandler project will be documented.

3.1 Statshandler

`StatsHandler.`**`setup_logging`**(*loglevel*, *set_streamhandler=True*)

Set up basic logging functionality.

This method sets up default logging facilities using the built in logging module. A default loglevel can be set, and the decision whether to log to a file switched on and of.

Parameters

- **`loglevel`** – The loglevel that will end up in the outputed logs
- **`set_streamhandler`** – If set, a file will be created named /var/log/statshandler/statshandler.log

`StatsHandler.`**`parseargs`**()

Parsing arguments passed

Using the built in argparse module, passed arguments are parsed. Simply calling this method is sufficient, the results will be returned.

Returns A namespace object containing all parsed arguments

`StatsHandler.`**`mainloop`**(*pubsub*, *redis_instance*, *redis_channel*, *gather_stats*, *analyzers*)

The main event loop

Parameters

- **`pubsub`** – The redis publish-subscribe object from setup_redis.
- **`redis_instance`** – The redis instance itself.
- **`redis_channel`** – The redis channel/key to publish to
- **`gather_stats`** – A boolean whether to gather statistics or not
- **`analyzers`** – A list of instantiated analyzers being derivatives from Analyzers.BaseAnalyzer

`StatsHandler.`**`main`**()

The main method.

First, arguments passed to the system are parsed. Then, logging is set up. The connection to redis is established by calling setup_redis() and the main event loop started.

Returns The system exit code

3.2 src.Monitoring

src.Monitoring.**analyze**(*statistics*, *analyzers*)
Analyzing the provided stats to alert on.

Parameters

- **statistics** – The data in a dictionary to analyze
- **analyzers** – A list of tripples in the following format: (analyzer: callable, arguments: list, filter: str)

Returns A list of alerts to publish

src.Monitoring.**collect_port_stats**(*statistics*)
Gathering statistics about OVS ports.

Parameters **statistics** – A dictionary to fetch statistics from.

Returns A list of statistics gathered.

src.Monitoring.**collect_portaggregate_stats**(*statistics*)
Gathering statistics on flows used on ports.

Parameters **statistics** – A dictionary to fetch statistics from.

Returns A list of statistics gathered.

src.Monitoring.**collect_stats**(*statistics*, *collectors*)
Applying a list of callable collectors to the statistics message.

Parameters

- **statistics** – A dictionary of received statistics.
- **collectors** – A list of callables in the format [(callable, (argument1, ...),filter), ...].

Returns A list of statistics gathered.

3.3 src.Analyzers

All available analyzers are shown here.

First, a base class is provided to give other analyzers a standardize whay of checking for setting up an alerting mechanism and generating alert messages.

class src.Analyzers.**BaseAnalyzer**(*alerter*, ***alerter_setup*)

__init__(*alerter*, ***alerter_setup*)
A basis for other analyzers to inherit from.

Providing three different alerting methods, namely a limit, a range and a standard deviation, all subclasses can use this reference implementation.

Parameters

- **alerter** – choose an alerting method like self._alert_limiter, ...For a chosen alerter, additional values have to be set. These are specified in their method documentation.
- **alerter_setup** – additional arguments needed to set up the alerter. These differ for alerters and are specified in their method documentation.

_alert_limiter(*value*)
Generating an alert by comparing to a simple limit.

Necessary to use this alerter is to specify a limit when instantiating the class by providing the constructor with limit=<n>.

As called, an alert is generated if value is smaller than self.limit. Internally, this reverts to the range alerter, as a limit can be expressed with a bottom value of - infinity and a top value of limit.

Parameters **value** – The newest value.

Returns True, if an alert is necessary.

_alert_range (*value*)

Generating an alert by comparing value to a given range of valid data.

Necessary to use this alerter is to specify a top and bottom limit when instantiating the class. This is done by providing the constructor with top=<n> and bottom=<m>; n and m have to be floats.

This alerts, if value is _not_ inside the range. If value is either supremum or infimum of the range, no alert is generated. For example, let value = 5.0. For bottom = 4.0 and top = 5.0, no alert is generated. If value changes to 5.1, an alert is generated.

Parameters **value** (*float*) – The new value to compare against.

Returns True, if value not in [bottom, ..., top]

_alert_standard_dev (*new_value*)

Generating an alert by checking against a range based on the standard deviation.

Necessary to use this alerter is to specify last_n values and a multiplier. Last_n must be an integer and is used to determine, how many last values to keep. A range is then determined by computing the mean of this n last values. Adding and subtracting the standard deviation of the list of last values multiplied by the set multiplier gives a range of mean +- (m * standard_dev). A new value is only added to the list of last values, if it does not generate an alert.

Parameters **new_value** – The newest value to check against.

Returns True, if new_value is not in mean +- (m * standard_dev).

analyze (*new_statistics*)

The analyze method to be overridden by all subclasses.

This method is called by the program logic, to get a list of alerts back if any are present. It _needs_ to be overriden in all subclasses.

Parameters **new_statistics** – The newest statistics gathered as a dictionary.

Returns A list of alerts.

generate_alert_message (*signature_id*, *additional_info={}*)

Base method to generate alert messages

A signature is the minimum necessary to be able to identify an alert. Additional information can be passed as a dictionary to be merged into the alert message returned.

Parameters

- **signature_id** – An ID to identify the alert.
- **additional_info** – A dictionary of additional information.

Returns A combined alert message ready to be used in the output phase.

class src.Analyzers.**PortsAnalyzer** (*limit*, *excluded_ports=[]*)

__init__ (*limit*, *excluded_ports=[]*)

This analyzer can handle single port statistics using a limit as alerter.

Parameters

- **limit** – The limit to alert on.
- **excluded_ports** – A list of ports not to produce alerts for.

analyze (*new_statistics*)
Single port statistics are analysed here.

For every switch and every port, statistics are gathered and alerts generated if a port surpasses the limit value with its rx_packets_delta. This is counting packets received by each port, even if no flow matched against them or a flow matched, but they were dropped.

Parameters **new_statistics** – The collected statistics on one Port.

class src.Analyzers.**PortAggregateAnalyzer** (*limit*)

__init__ (*limit*)
This analyzer can handler aggregated statistics combining port and flow information based on a limiter.

Parameters **limit** – The limit to alert on.

analyze (*new_statistics*)
Aggregated port statistics about used flows per port are analysed here.

For every switch and every port on them the limit value is checked against the flow_count usage of this port.

Parameters **new_statistics** – The dictionary containing raw statistics.

3.4 src.redis

src.redis.**setup_redis** (*ip*, *port*, *key*, *max_retries=30*, *retry=0*)
This will connect to a Redis instance and return the connection object.

Establishing a stable connection with a Redis instance is the goal. Therefore, the method will retry at most max_retries times.

Parameters

- **ip** – The IP of the Redis server to connect to.
- **port** – The port on which Redis is running.
- **key** – The redis channel/key to subscribe the a PubSub object to.
- **max_retries** – The maximum number of retries after which connection attempts will fail.
- **retry** – The current number of retries executed.

Returns A set of (redis_instance, pubsub)

src.redis.**_fetch_message** (*pubsub*)
Fetching a single message from redis.

This fetches a single message and ensures, it has a valid data field.

Parameters **pubsub** – The pubsub object created by setup_redis to fetch the message from.

Returns A complete and data filled message.

Return type Python dict

src.redis.**_parse_message** (*message*)
Parsing a message to get its valuable parts.

A message received from Redis contains information in json format and metadata from Redis, extracting the data is done here.

Parameters **message** – Raw message received from redis.

Returns A dictionary containing parsed data `src.redis.`**`get_statistics`**(*pubsub*)

Returning a fully processed Redis message.

Parameters **`pubsub`** – The pubsub object created by setup_redis to fetch the message from.

Returns A received and parsed message.

`src.redis.`**`publish_messages`**(*messages*, *redis_instance*, *channel*)

Publish a list of messages via Redis.

Parameters

- **`messages`** – A list of json-dumpable messages to send.
- **`redis_instance`** – The instance to publish the messages to.
- **`channel`** – The Redis channel to publish to.

Returns

PYTHON MODULE INDEX

S

D. SynFlood Documentation

SynFlood Documentation

Release 1.0.0rc0

Christoph Girstenbrei

Dec 09, 2016

CONTENTS:

CHAPTER

ONE

USAGE DOCUMENTATION

This script is intended to be used as a testing tool to perform SYN flood attacks against a known host.

1.1 Requirements

Required to run this script is at least Python 3.4.

1.2 Usage

The script can be run in standalone mode using the following options:

- -l : Setting the loglevel to one of {critical,warning,error,debug,info}
- -s : The source IP address to use or 'r' for random to spoof the address
- -t : The target IP address to attack
- -p : The source TCP port to use or 'r' for random to switch TCP port for every connection
- -P : The destination TCP port to use or 'r' for random to switch TCP port for every connection
- -n : The number of packages to send to the target
- -w : The wait period in between packages in milliseconds

To attack a <target> IP with 1000000 packages from different TCP source ports to TCP port 80, the following can be used: ./SynFlood.py -s r -t <target> -n 1000000 -p r -P 80

CHAPTER

TWO

DOCUMENTED MODULES

In the following sections, the modules contained within the SynFlood project will be documented.

2.1 SynFlood

This software is based on a script developed by Silver Moon. The following is the original copyright message:

Syn flood program in python using raw sockets (Linux)

Silver Moon (m00n.silv3r@gmail.com) source is http://www.binarytides.com/python-syn-flood-program-raw-sockets-linux/

`SynFlood.`**`parseargs`**`()`

Parsing arguments passed

Using the built in argparse module, passed arguments are parsed. Simply calling this method is sufficient, the results will be returned.

Returns A namespace object containing all parsed arguments

PYTHON MODULE INDEX

S

E. FlowFlood Documentation

FlowFlood Documentation

Release 1.0.0rc0

Christoph Girstenbrei

Dec 09, 2016

CONTENTS:

CHAPTER

ONE

USAGE DOCUMENTATION

This script is intended to perform attacks in an SDN environment. It generates Flows in a MAC-learning SDN switch by performing a TCP handshake and sending a package back and forth.

1.1 Requirements

Required to run this script is at least Python 3.4.

1.2 Usage

The script can be run in standalone mode using the following options:

- -l : Setting a loglevel from one of {info,debug,error,critical,warning}
- -m : Running in master mode.

The simplest setup is having one instance running with -m as master and one without.

CHAPTER

TWO

DOCUMENTED MODULES

In the following sections, the modules contained within the Statshandler project will be documented.

2.1 FlowFlood

`FlowFlood.`**`setup_logging`**(*loglevel*, *set_streamhandler=True*)

Set up basic logging functionality.

This method sets up default logging facilities using the built in logging module. A default loglevel can be set, and the decision whether to log to a file switched on and of.

Parameters

- **`loglevel`** – The loglevel that will end up in the outputed logs
- **`set_streamhandler`** – If set, a file will be created named /var/log/statshandler/statshandler.log

`FlowFlood.`**`parseargs`**()

Parsing arguments passed

Using the built in argparse module, passed arguments are parsed. Simply calling this method is sufficient, the results will be returned.

Returns A namespace object containing all parsed arguments

`FlowFlood.`**`mainloop`**(*args*)

The main event loop.

If this is running as the master, the MAC address is changed, a socket set up and a connection is waited for. As a connection is established, one package is received and one sent.

If this is running in slave mode, connection to master is attempted. On success, a package is sent and one received.

Parameters **`args`** – Argumentes passed to the script.

`FlowFlood.`**`main`**()

The main method.

First, arguments passed to the system are parsed. Then, logging is set up. The mainloop is then started, to run until keyboard interrupt.

Returns The system exit code

2.2 src.netio

`src.netio.`**`setup_socket`**`()`
Creating a TCP socket.

Returns A TCP socket.

`src.netio.`**`rand_mac`**`()`
Generating a random MAC address in private MAC address space.

Returns The random MAC as a string.

`src.netio.`**`change_hwaddr`**`(`*interface='eth1'*`)`
Changing the MAC on interface.

Parameters **`interface`** – The one to change the MAC on.

`src.netio.`**`setup_socket_master`**`(`*src_ip*`)`
Setting up a server socket.

This sets up a server socket listening on address src_ip and a random port.

Parameters **`src_ip`** – The ip to bind to.

Returns A server socket.

`src.netio.`**`yld_port`**`()`
To generate a series of random ports, this returns a generator.

Returns A generator object returning ints from 1025 to 32768

`src.netio.`**`next_port`**`()`
Giving back the next port as an integer.

Returns Integer form 1025 to 32768.

`src.netio.`**`stable_connect`**`(`*sock*, *dst_ip*, *dst_port*`)`
Establishing a tcp connection to dst_ip.

This method tries to connect to a TCP socket, catching many exceptions possibly occuring during this operations.

Parameters

- **`sock`** – The socket to use to connect via.
- **`dst_ip`** – The IP to connect to.
- **`dst_port`** – The port to connect to.

`src.netio.`**`clear_arp`**`(`*ip*`)`
Clearing the arp cache of the host.

Parameters **`ip`** – IP that gets deleted

Returns The return code of the arp clearing command

PYTHON MODULE INDEX

F. VM Vagrant File

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The "2" in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure(2) do |config|
 # The most common configuration options are documented and commented below.
 # For a complete reference, please see the online documentation at
 # https://docs.vagrantup.com.

 # Every Vagrant development environment requires a box. You can search for
 # boxes at https://atlas.hashicorp.com/search.
 config.vm.box = "bento/ubuntu-16.04"
 config.vm.box_version = "2.2.9"
 #config.vm.synced_folder ".", "/vagrant"

 # Disable automatic box update checking. If you disable this, then
 # boxes will only be checked for updates when the user runs
 # `vagrant box outdated`. This is not recommended.
 config.vm.box_check_update = false

 # Create a forwarded port mapping which allows access to a specific port
 # within the machine from a port on the host machine. In the example below,
 # accessing "localhost:8080" will access port 80 on the guest machine.
 #config.vm.network "forwarded_port", guest: 80, host: 8080
 #config.vm.network "forwarded_port", guest: 81, host: 8081

 # Create a private network, which allows host-only access to the machine
 # using a specific IP.
 config.vm.network "private_network", ip: "192.168.33.10"

 # Create a public network, which generally matched to bridged network.
 # Bridged networks make the machine appear as another physical device on
 # your network.
 # config.vm.network "public_network"

 # Share an additional folder to the guest VM. The first argument is
 # the path on the host to the actual folder. The second argument is
 # the path on the guest to mount the folder. And the optional third
 # argument is a set of non-required options.
 # config.vm.synced_folder "../data", "/vagrant_data"
 config.vm.synced_folder ".", "/vagrant"

 # Provider-specific configuration so you can fine-tune various
 # backing providers for Vagrant. These expose provider-specific options.
 # Example for VirtualBox:
 #
config.vm.provider "virtualbox" do |vb|
 # Display the VirtualBox GUI when booting the machine
 # vb.gui = true
 #
 #  # Customize the amount of memory on the VM:
 vb.cpus = 12
 vb.memory = "8192"
```

```
end
 #
 # View the documentation for the provider you are using for more
 # information on available options.

 # Define a Vagrant Push strategy for pushing to Atlas. Other push strategies
 # such as FTP and Heroku are also available. See the documentation at
 # https://docs.vagrantup.com/v2/push/atlas.html for more information.
 # config.push.define "atlas" do |push|
 #   push.app = "YOUR_ATLAS_USERNAME/YOUR_APPLICATION_NAME"
 # end

 # Enable provisioning with a shell script. Additional provisioners such as
 # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
 # documentation for more information about their specific syntax and use.
 config.vm.provision "shell", inline: <<-SHELL

  PURPLE='\033[0;35m'
  NC='\033[0m' # No Color
  # Update system
  printf "${PURPLE}|----------   Updating         ---------|${NC}"

  sudo timedatectl set-timezone Europe/Berlin
  sudo cp /vagrant/sources_pre.list /etc/apt/sources.list
  sudo apt-mark hold grub-pc
  sudo apt-get update
  sudo apt-get upgrade -y
  sudo apt-get install -y -q tmux vim htop tofrodos curl
  sudo apt-get install -y -q lxc

  # Install OpenvSwitch
  sudo apt-get install -y -q openvswitch-switch

  # Install Vagrant
  printf "${PURPLE}|----------   Installing Vagrant    ---------|${NC}"
  /vagrant/load_vagrant.sh
  sudo dpkg -i vagrant_1.8.5_x86_64.deb
  sudo vagrant plugin install vagrant-lxc

  # Loading Cache
  /vagrant/load_cache.sh

  # Adding bridge to ovs
  printf "${PURPLE}|----------   Adding ovs bridge     ---------|${NC}"
  sudo ovs-vsctl add-br switch0
  sudo ovs-vsctl add-br switch1
  sudo ovs-vsctl add-br switch-private

  #Create Patch between switch0 and switch1
  printf "${PURPLE}|----------   Patching switches     ---------|${NC}"
  sudo ovs-vsctl add-port switch0 patch0-1 -- set interface patch0-1 type=patch
    options:peer=patch1-0
  sudo ovs-vsctl add-port switch1 patch1-0 -- set interface patch1-0 type=patch
    options:peer=patch0-1

  # Adding veth-pair
  printf "${PURPLE}|----------   Adding veth-pair     ---------|${NC}"
  ip link add ovs-acc type veth peer name ovs-ins
  ip link add ovs-acc-priv type veth peer name ovs-ins-priv

  # Setting up outside port
  printf "${PURPLE}|----------   Outside Port       ---------|${NC}"
  ifconfig ovs-acc 10.10.10.10/16
  ifconfig ovs-acc-priv 10.20.0.10/16
  ifconfig ovs-acc up
  ifconfig ovs-acc-priv up
```

```
# Setting up inside port
printf "${PURPLE}|----------   Inside Port        ---------|${NC}"
ovs-vsctl add-port switch0 ovs-ins
ovs-vsctl add-port switch-private ovs-ins-priv
ifconfig ovs-ins up
ifconfig ovs-ins-priv up

# Allow access from internal network to external
printf "${PURPLE}|----------   IpTables -> Inet     ---------|${NC}"
sudo iptables -t nat -A POSTROUTING -o enp0s3 -j MASQUERADE

#Connect eth0 to port and configure dhclient
printf "${PURPLE}|----------   Set OVS Version      ---------|${NC}"
ovs-vsctl set bridge switch0
  protocols=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13,OpenFlow14,OpenFlow15
ovs-vsctl set bridge switch1
  protocols=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13,OpenFlow14,OpenFlow15
ovs-vsctl set bridge switch-private
  protocols=OpenFlow10,OpenFlow11,OpenFlow12,OpenFlow13,OpenFlow14,OpenFlow15


# Copying the lxc-configs
printf "${PURPLE}|---------- Copy lxc configs via unison ---------|${NC}"
mkdir lxcs
cp -r /vagrant/lxcs/* lxcs

# Copy files to unix
printf "${PURPLE}|----------  Chmod files        ---------|${NC}"
sudo su
cd /home/vagrant/lxcs/
chmod +x ifup ifdown lxc.conf ifup1 ifdown1
chmod +x /home/vagrant/lxcs/

printf "${PURPLE}|----------   Converting files     ---------|${NC}"
fromdos -v ifup ifdown lxc.conf ifup1 ifdown1 ifup-private ifdown-private
  test-main.sh main/resolv.conf
find . -name "*.sh" | xargs fromdos -v
find . -name "*.yaml" | xargs fromdos -v
find . -name "*.rules" | xargs fromdos -v
cp lxc.conf /etc/init/

# Starting Vagrant container
printf "${PURPLE}|----------   Starting container    ---------|${NC}"
cd /home/vagrant/lxcs/main
sudo vagrant up dns redis
sudo vagrant up

# Forward Ports
printf "${PURPLE}|----------   Forward Ports       ---------|${NC}"

ip='10.20.0.9'
# Grafana
iptables -t nat -A PREROUTING -i enp0s8 -p tcp --dport 80 -j DNAT --to $ip:3000
# Kibana
iptables -t nat -A PREROUTING -i enp0s8 -p tcp --dport 81 -j DNAT --to $ip:80
# Flooding Port
iptables -t nat -A PREROUTING -i enp0s8 -p tcp --dport 1234 -j DNAT --to
  10.10.10.4:80

# Setting controller
printf "${PURPLE}|----------   Setting OVS controller  ---------|${NC}"
sudo ovs-vsctl set-controller switch0 tcp:10.10.10.8:6633
sudo ovs-vsctl set-controller switch1 tcp:10.10.10.8:6633

# Install monitoring
```

```
  wget -qO - https://artifacts.elastic.co/GPG-KEY-elasticsearch | sudo apt-key add -
  sudo apt-get install -y apt-transport-https
  echo "deb https://artifacts.elastic.co/packages/5.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-5.x.list
  sudo apt-get update && sudo apt-get install metricbeat
  cp /vagrant/lxcs/main/metricbeat.yml /etc/metricbeat/
  systemctl start metricbeat
  systemctl enable metricbeat

  # Disable Standalone mode
  #sudo ovs-vsctl set-fail-mode switch0 secure
  #sudo ovs-vsctl set-fail-mode switch1 secure

  # Limit the size of the flow table
  sudo ovs-vsctl -- --id=@ft create Flow_Table flow_limit=200 overflow_policy=refuse -- set Bridge switch0 flow_tables=0=@ft
  sudo ovs-vsctl -- --id=@ft create Flow_Table flow_limit=200 overflow_policy=refuse -- set Bridge switch1 flow_tables=0=@ft

  # Deactivating default LXC-bridge
  sudo ifconfig lxcbr0 down
  #sudo brctl delbr lxcbr0

  # Activating file sync
  sudo chmod a+rx /home/vagrant/lxcs/sync-files.sh
  sudo cp /home/vagrant/lxcs/sync-files.service /etc/systemd/system/
  sudo systemctl daemon-reload
  sudo systemctl enable sync-files
  sudo systemctl start sync-files

  # Deactivating syn cookies
  sudo sysctl -w net.ipv4.tcp_syncookies=0
  sudo systemctl restart networking

  # Saving Cache
  /vagrant/save_cache.sh

  /home/vagrant/lxcs/test-main.sh

  printf "Machine deployed."

 SHELL
end
```

G. LXCs Vagrant File

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# IP-Table #
# DNS         | 10.0.3.61, 10.10.10.2
# attackone    | 10.0.3.68, 10.10.10.3
# Manager     | 10.0.3.14, 10.10.10.8, 10.20.0.8
# Mitigation   | 10.0.3.187, 10.20.0.7
# Monitoring   | 10.0.3.119, 10.10.10.5, 10.20.0.5
# Redis       | 10.0.3.245, 10.20.0.6
# Target      | 10.0.3.162, 10.10.10.4
# ELK         | 10.20.0.9



Vagrant.configure("2") do |config|

  config.vm.define "dns" do |dns|
    dns.vm.hostname = "dns"
    dns.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    dns.vm.provider :lxc do |lxc, override|
      lxc.container_name = "dns"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup1'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown1'
      lxc.customize 'network.veth.pair', 'dns1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.10.10.2/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '0'
    end

    dns.vm.provision "shell", inline: <<-SHELL

      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      ip route de default
      ip route add default via 10.10.10.10
      ping -c 2 8.8.8.8
      ping -c 2 www.google.de
      sudo apt-get update
      sudo apt-get install -y bind9 dnsutils
      sudo sed -i 's|nameserver 8.8.8.8|nameserver 10.10.10.2|' /etc/resolv.conf
      sudo cp -f /vagrant/dns/named.conf.options /etc/bind
      sudo cp -f /vagrant/dns/named.conf.local /etc/bind
      sudo mkdir /etc/bind/zones
      sudo cp /vagrant/dns/db.sdn.local /etc/bind/zones
      sudo cp /vagrant/dns/db.10.10.10 /etc/bind/zones
      sudo systemctl start bind9
    SHELL
  end

```

```
config.vm.define "target" do |target|
  target.vm.hostname = "target"
  target.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

  target.vm.provider :lxc do |lxc, override|
    lxc.container_name = "target"
    lxc.customize 'start.auto', '1'
    lxc.customize 'start.delay', '3'
    lxc.customize 'network.type', 'veth'
    lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup1'
    lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown1'
    lxc.customize 'network.veth.pair', 'target1eth1'
    lxc.customize 'network.flags', 'up'
    lxc.customize 'network.ipv4', '10.10.10.4/16'
    lxc.customize 'group', 'onboot'
    lxc.customize 'cgroup.cpuset.cpus', '1'
  end

  target.vm.provision "shell", inline: <<-SHELL

    # Custom configuration here!
    sudo timedatectl set-timezone Europe/Berlin
    sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
    ip route de default
    ip route add default via 10.10.10.10
    ping -c 2 8.8.8.8
    ping -c 2 www.google.de
    sudo apt-get install -y -q tcpdump iperf
    cp /vagrant/resolv.conf /etc
    sudo apt-get update
    sudo apt-get install -y nginx
    sudo systemctl enable nginx
    sudo systemctl start nginx
    sudo cp /vagrant/target/BA.mp4 /var/www/html
    sudo cp /vagrant/target/index.nginx-debian.html /var/www/html

  SHELL
end

config.vm.define "attackone" do |attackone|
  attackone.vm.hostname = "attackone"
  attackone.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

  attackone.vm.provider :lxc do |lxc, override|
    lxc.container_name = "attackone"
    lxc.customize 'start.auto', '1'
    lxc.customize 'start.delay', '3'
    lxc.customize 'network.type', 'veth'
    lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup'
    lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown'
    lxc.customize 'network.veth.pair', 'attackone1eth1'
    lxc.customize 'network.flags', 'up'
    lxc.customize 'network.ipv4', '10.10.10.3/16'
    lxc.customize 'group', 'onboot'
    lxc.customize 'cgroup.cpuset.cpus', '2'
  end

  attackone.vm.provision "shell", inline: <<-SHELL

    # Custom configuration here!
    sudo timedatectl set-timezone Europe/Berlin
    sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
    ip route de default
    ip route add default via 10.10.10.10
    ping -c 2 8.8.8.8
    ping -c 2 www.google.de
```

```
      sudo apt-get install -y -q iperf iptables tcpdump ethtool
      cp /vagrant/resolv.conf /etc

      #sudo ethtool -K eth1 gro off
      #sudo ifconfig eth1 promisc

    SHELL
  end

  config.vm.define "attacktwo" do |attacktwo|
    attacktwo.vm.hostname = "attacktwo"
    attacktwo.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    attacktwo.vm.provider :lxc do |lxc, override|
      lxc.container_name = "attacktwo"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown'
      lxc.customize 'network.veth.pair', 'attacktwo1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.10.10.6/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '3'
    end

    attacktwo.vm.provision "shell", inline: <<-SHELL

      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      ip route de default
      ip route add default via 10.10.10.10
      ping -c 2 8.8.8.8
      ping -c 2 www.google.de
      sudo apt-get install -y -q iperf iptables tcpdump ethtool
      cp /vagrant/resolv.conf /etc

      #sudo ethtool -K eth1 gro off
      #sudo ifconfig eth1 promisc
    SHELL
  end

  config.vm.define "monitoring" do |monitoring|
    monitoring.vm.hostname = "monitoring"
    monitoring.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    monitoring.vm.provider :lxc do |lxc, override|
      lxc.container_name = "monitoring"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown'
      lxc.customize 'network.veth.pair', 'monit1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.10.10.5/16'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup-private'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown-private'
      lxc.customize 'network.veth.pair', 'monit1eth2'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.20.0.5/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '4'
```

```
    end

    monitoring.vm.provision "shell", inline: <<-SHELL
      PURPLE='\033[0;35m'
      NC='\033[0m' # No Color
      # Standard Base Config
      printf "${PURPLE}|---------- Monitoring: Base setup   ---------|${NC}"
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      sudo ip link set down eth2 && sudo ip link set up eth2
      ip route del default && ip route add default via 10.20.0.10
      sudo apt-get install -y iperf
      sudo apt-get update
      sudo apt-get install -y ethtool
      sudo apt-get install -y python3-pip
      sudo pip3 install --upgrade pip
      sudo pip3 install -r /vagrant/monitoring/statistical/requirements.txt

      # Configure for packet sniffing
      printf "${PURPLE}|---------- Monitoring: Promisc    ---------|${NC}"
      sudo ethtool -K eth1 gro off
      sudo ifconfig eth1 promisc

      # Install & configure Suricata
      printf "${PURPLE}|---------- Monitoring: Suricata    ---------|${NC}"
      apt install -y software-properties-common
      sudo add-apt-repository -y ppa:oisf/suricata-stable
      sudo apt-get update
      sudo apt-get -y install suricata
      sudo systemctl stop suricata
      sudo cp /vagrant/monitoring/local.rules /etc/suricata/rules/
      sudo cp /vagrant/monitoring/suricata.yaml /etc/suricata/
      sudo sed -i 's/IFACE=eth0/IFACE=eth1/' /etc/default/suricata
      sudo sed -i 's/LISTENMODE=af-packet/LISTENMODE=pcap/' /etc/default/suricata

      sleep 2
      sudo systemctl enable suricata
      sudo systemctl start suricata

      sleep 2

      systemctl restart suricata

      cp /vagrant/monitoring/statistical/statshandler.service /etc/systemd/system/
      systemctl daemon-reload
      systemctl enable statshandler
      systemctl start statshandler
      cp /vagrant/resolv.conf /etc
    SHELL
  end

  config.vm.define "redis" do |redis|
    redis.vm.hostname = "redis"
    redis.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    redis.vm.provider :lxc do |lxc, override|
      lxc.container_name = "redis"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup-private'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown-private'
      lxc.customize 'network.veth.pair', 'redis1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.20.0.6/16'
      lxc.customize 'group', 'onboot'
```

```
      lxc.customize 'cgroup.cpuset.cpus', '5'
    end

    redis.vm.provision "shell", inline: <<-SHELL
      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      sudo ip link set down eth1 && sudo ip link set up eth1
      ip route del default && ip route add default via 10.20.0.10

      /vagrant/redis/setup-redis.sh


    SHELL
  end

  config.vm.define "elk" do |elk|
    elk.vm.hostname = "elk"
    elk.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    elk.vm.provider :lxc do |lxc, override|
      lxc.container_name = "elk"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup-private'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown-private'
      lxc.customize 'network.veth.pair', 'elk1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.20.0.9/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '6'
    end

    elk.vm.provision "shell", inline: <<-SHELL
      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      sudo ip link set down eth1 && sudo ip link set up eth1
      ip route del default && ip route add default via 10.20.0.10

      printf "${PURPLE}|---------- Monitoring: Copy elk setup ---------|${NC}"
      cd /home/vagrant/
      cp /vagrant/elk/*.sh ./
      # Make files executable
      sudo chmod +x *.sh

      # Install & configure elk stack
      printf "${PURPLE}|---------- Monitoring: ELK setup   ---------|${NC}"
      sudo ./setup-elk.sh
      sudo ./configure-grafana.sh

    SHELL
  end

  config.vm.define "manager" do |manager|
    manager.vm.hostname = "manager"
    manager.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    manager.vm.provider :lxc do |lxc, override|
      lxc.container_name = "manager"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown'
```

```
      lxc.customize 'network.veth.pair', 'mane1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.10.10.8/16'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup-private'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown-private'
      lxc.customize 'network.veth.pair', 'mane1eth2'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.20.0.8/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '7'
    end

    manager.vm.provision "shell", inline: <<-SHELL

      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      sudo ip link set down eth1 && sudo ip link set up eth1
      ip route del default && ip route add default via 10.20.0.10
      echo "install iperf"
      sudo apt-get install -y iperf
      echo "Installing Ryu..."
      LC_ALL=C
      sudo apt-get update
      sudo apt-get install -y python3-dev
      sudo apt-get install -y python3-pip
      sudo apt-get install -y python3-eventlet
      sudo apt-get install -y python3-routes
      sudo apt-get install -y python3-webob
      sudo apt-get install -y python3-paramiko
      sudo pip3 install --upgrade pip
      sudo pip3 install --upgrade six
      sudo pip3 install redis
      sudo pip3 install hiredis
      sudo pip3 install ryu
      sudo pip3 install --upgrade tinyrpc
      cp /vagrant/manager/controller.py ./

      cp /vagrant/manager/controller.service /etc/systemd/system/
      systemctl daemon-reload
      systemctl enable controller
      systemctl start controller
      cp /vagrant/resolv.conf /etc
    SHELL
  end

  config.vm.define "mitigation" do |mitigation|
    mitigation.vm.hostname = "mitigation"
    mitigation.vm.box = "developerinlondon/ubuntu_lxc_xenial_x64"

    mitigation.vm.provider :lxc do |lxc, override|
      lxc.container_name = "mitigation"
      lxc.customize 'start.auto', '1'
      lxc.customize 'start.delay', '3'
      lxc.customize 'network.type', 'veth'
      lxc.customize 'network.script.up', '/home/vagrant/lxcs/ifup-private'
      lxc.customize 'network.script.down', '/home/vagrant/lxcs/ifdown-private'
      lxc.customize 'network.veth.pair', 'mit1eth1'
      lxc.customize 'network.flags', 'up'
      lxc.customize 'network.ipv4', '10.20.0.7/16'
      lxc.customize 'group', 'onboot'
      lxc.customize 'cgroup.cpuset.cpus', '8'
    end

    mitigation.vm.provision "shell", inline: <<-SHELL
```

```
      # Custom configuration here!
      sudo timedatectl set-timezone Europe/Berlin
      sudo sed -i 's|nameserver 10.0.3.1|nameserver 8.8.8.8|' /etc/resolv.conf
      sudo ip link set down eth1 && sudo ip link set up eth1
      ip route del default && ip route add default via 10.20.0.10

      sudo apt-get update
      sudo apt-get install -y iperf
      sudo apt-get install -y python3-pip
      sudo pip3 install --upgrade pip
      sudo pip3 install --upgrade requests hiredis redis

      cp /vagrant/mitigation/miti*.service /etc/systemd/system/
      systemctl daemon-reload
      systemctl enable miti-stats
      systemctl start miti-stats
      systemctl enable miti-ids
      systemctl start miti-ids
      systemctl enable miti-mit
      systemctl start miti-mit
    SHELL
  end
end
```

Acronyms

CPU	Central Processing Unit
TLS	Transport Layer Security
DNS	Domain Name System
DoS	Denial Of Service
DDoS	Distributed Denial of Service
EVE	Extensible Event Format
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDMEF	Intrusion Detection Message Exchange Format
IDS	Intrusion Detection System
IP	Internet Protocol
JSON	JavaScript Object Notation
LXC	Linux Container
MAC	Media-Access-Control
NIC	Network Interface Card
NOS	Network Operating System
OF	OpenFlow
OVS	OpenVSwitch
SDN	Software Defined Network
SNMP	Simple Network Management Protocol
SRP	Single Responsibility Principle
TCP	Transport Control Protocol
TES	Triple Exponential Smoothing
TLS	Transport Layer Security
VM	Virtual Machine
XML	Extensible Markup Language

List of Figures

List of Tables

Bibliography

[Amb+15] Moreno Ambrosin et al. "Lineswitch: Efficiently managing switch flow in software-defined networking while effectively tackling dos attacks". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM. 2015, pp. 639–644.

[AX15] Izzat Alsmadi and Dianxiang Xu. "Security of software defined networks: A survey". In: *computers & security* 53 (2015), pp. 79–108.

[BAM09] Theophilus Benson, Aditya Akella, and David A Maltz. "Unraveling the Complexity of Network Management." In: *NSDI*. 2009, pp. 335–348.

[BBC14] BBC. *Sony Pictures computer system hacked in online attack*. 2014. URL: `http://www.bbc.com/news/technology-30189029` (visited on 11/21/2016).

[BCS13] Kevin Benton, L Jean Camp, and Chris Small. "Openflow vulnerability assessment". In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM. 2013, pp. 151–152.

[BG13] Marcelo Bagnulo and Alberto Garcia-Martinez. "SAVI: The IETF standard in address validation". In: *IEEE Communications Magazine* 51.4 (2013), pp. 66–73.

[Cho+14] S. R. Chowdhury et al. "PayLess: A low cost network monitoring framework for Software Defined Networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. May 2014, pp. 1–9. DOI: `10.1109/NOMS.2014.6838227`.

[Cui+16] Yunhe Cui et al. "SD-Anti-DDoS: Fast and efficient DDoS defense in software-defined networks". In: *Journal of Network and Computer Applications* 68 (2016), pp. 65–79.

[DBP05] Thomas Dubendorfer, Matthias Bossardt, and Bernhard Plattner. "Adaptive distributed traffic control service for DDoS attack mitigation". In: *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2005, 8–pp.

[Deb+07] H. Debar et al. *The Intrusion Detection Message Exchange Format*. Language. Online. RFC. Internet Engineering Steering Group, Mar. 2007. URL: `https://tools.ietf.org/html/rfc4765`.

[Edd06] Wesley M Eddy. "Defenses against TCP SYN flooding attacks". In: *The Internet Protocol Journal* 9.4 (2006), pp. 2–16.

[Fil+13] James J. Filliben et al. *Engineering Statistics Handbook*. National Institute of Standards and Technology. Oct. 2013. URL: `http://www.itl.nist.gov/div898/handbook//index.htm`.

[GB14] Paul Goransson and Chuck Black. *Software Defined Networks: A Comprehensive Approach*. Elsevier, 2014.

[GDK] Tobias Guggemos, Vitalian Danciu, and Dieter Kranzlmüller. “Schichtung virtueller Maschinen zu Labor-und Lehrinfrastruktur”. In: *Gesellschaft für Informatik eV (GI) publishes this series in order to make available to a broad public recent findings in informatics (ie computer science and informa-tion systems), to document conferences that are organized in co-operation with GI and to publish the annual GI Award dissertation.* P. 35.

[Gio+14] Kostas Giotis et al. “Combining OpenFlow and sFlow for an effective and scalable anomaly detection and mitigation mechanism on SDN environments”. In: *Computer Networks* 62 (2014), pp. 122–136.

[Gua16] The Guardian. *Major cyber attack disrupts internet service across Europe and US.* 2016. URL: `https://www.theguardian.com/technology/2016/oct/21/ddos-attack-dyn-internet-denial-service` (visited on 11/21/2016).

[Hab14] Itamar Haber. *Using stunnel to Secure Redis.* English. Mar. 2014. URL: `https://redislabs.com/blog/using-stunnel-to-secure-redis`.

[Her+16] Andreas Herz et al. *Suricata User Guide.* OISF. 2016. URL: `https://suricata.readthedocs.io/en/latest/rules/index.html`.

[Hon+15] Sungmin Hong et al. “Poisoning Network Visibility in Software-Defined Networks: New Attacks and Countermeasures.” In: *NDSS.* 2015.

[Hu+14] Hongxin Hu et al. “FLOWGUARD: building robust firewalls for software-defined networks”. In: *Proceedings of the third workshop on Hot topics in software defined networking.* ACM. 2014, pp. 97–102.

[ISO] 'ISO/IEC'. *Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model.* ISO/IOC. URL: `http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip`.

[JM16] Damian Janowski and Michel Martens. *redis.io.* English. 2016. URL: `http://redis.io/`.

[JMD14] Yosr Jarraya, Taous Madi, and Mourad Debbabi. “A survey and a layered taxonomy of software-defined networking”. In: *Communications Surveys & Tutorials, IEEE* 16.4 (2014), pp. 1955–1980.

[KKS13] Rowan Klöti, Vasileios Kotronis, and Paul Smith. “Openflow: A security analysis”. In: *2013 21st IEEE International Conference on Network Protocols (ICNP).* IEEE. 2013, pp. 1–6.

[Kre+15] Diego Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76.

[Lau+00] Felix Lau et al. “Distributed denial of service attacks”. In: *Systems, Man, and Cybernetics, 2000 IEEE International Conference on.* Vol. 3. IEEE. 2000, pp. 2275–2280.

[Lem+02] Jonathan Lemon et al. “Resisting SYN Flood DoS Attacks with a SYN Cache.” In: *BSDCon.* Vol. 2002. 2002, pp. 89–97.

[LMK16] Wenjuan Li, Weizhi Meng, and Lam For Kwok. “A survey on OpenFlow-based Software Defined Networks: Security challenges and countermeasures”. In: *Journal of Network and Computer Applications* 68 (2016), pp. 126–139.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[Mar16] Alex Marczinek. *Mitigation of attacks in the environment of Software Defined Networks.* PDF. Dec. 2016.

[MBR16] Louis Marinos, Adrian Belmonte, and Evangelos Rekleitis. *ENISA Threat Landscape 2015.* 2016.

[McK+08] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[MR04] Jelena Mirkovic and Peter Reiher. "A taxonomy of DDoS attack and DDoS defense mechanisms". In: *ACM SIGCOMM Computer Communication Review* 34.2 (2004), pp. 39–53.

[NG13] Thomas D Nadeau and Ken Gray. *SDN: software defined networks.* " O'Reilly Media, Inc.", 2013.

[Nyg+14] Anders Nygren et al. *OpenFlow Switch Specification Version 1.5.0 (Protocol version 0x06).* PDF. Dec. 2014. URL: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`.

[ONF14] ONF. *SDN Architecture Overview.* Tech. rep. Open Networking Foundation (ONF), Nov. 2014.

[Sec16] Calyptix Security. *Top 7 Network Attack Types in 2016.* 2016. URL: `http://www.calyptix.com/top-threats/top-7-network-attack-types-2016/` (visited on 08/10/2016).

[She+12] Justine Sherry et al. "Making middleboxes someone else's problem: network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[Shi+13] Seungwon Shin et al. "AVANT-GUARD: scalable and vigilant switch flow management in software-defined networks". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security.* ACM. 2013, pp. 413–424.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. ISSN: 0018-8670. DOI: `10.1147/sj.132.0115`.

[Sys15] CS Communication & Systems. *Prelude OSS — The Open Source Reference.* Online. 2015. URL: `http://www.prelude-siem.com/en/products/prelude-oss/`.

[Tan03] Matthew Tanase. "IP spoofing: an introduction". In: *Security Focus* 11 (2003).

[Tea00] Nagios Plugins Team. *Nagios Plugin Development Guidelines.* Nagios Enterprises. 2000. URL: `https://nagios-plugins.org/doc/guidelines.html`.

[VE06] Randal Vaughn and Gadi Evron. "DNS amplification attacks". In: *Go online to http://www. isotf. org/news/DNS-Amplification-Attacks. pdf* (2006).

[Ver16] Verisign. *Q3 2016 DDoS trends report: UDP Flood Attacks make up 49 percent of attacks*. 2016. URL: `https://blog.verisign.com/security/q3-2016-ddos-trends-report-udp-flood-attacks-make-up-49-percent-of-attacks/` (visited on 11/26/2016).

[YBX11] Guang Yao, Jun Bi, and Peiyao Xiao. "Source address validation solution with OpenFlow/NOX architecture". In: *2011 19th IEEE International Conference on Network Protocols*. IEEE. 2011, pp. 7–12.

[ZJT13] Saman Taghavi Zargar, James Joshi, and David Tipper. "A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks". In: *IEEE Communications Surveys & Tutorials* 15.4 (2013), pp. 2046–2069.