ANALYSIS OF DENIAL-OF-SERVICE ATTACK VECTORS IN SOFTWARE-DEFINED NETWORKS

THESIS

Anthony A. Portante, 2nd Lieutenant, USAF

AFIT-ENG-MS-17-M-060

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-17-M-060

ANALYSIS OF DENIAL-OF-SERVICE ATTACK VECTORS IN SOFTWARE-DEFINED NETWORKS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

Anthony A. Portante, B.S.C.S.

2nd Lieutenant, USAF

March 2017

AFIT-ENG-MS-17-M-060

ANALYSIS OF DENIAL-OF-SERVICE ATTACK VECTORS IN SOFTWARE-DEFINED NETWORKS

THESIS

Anthony A. Portante, B.S.C.S.
2nd Lieutenant, USAF

Committee Membership:

Barry E. Mullins, Ph.D., P.E.
Chair

Timothy H. Lacey, Ph.D., CISSP
Member

Michael R. Grimaila, Ph.D., CISM, CISSP
Member

AFIT-ENG-MS-17-M-060

Abstract

Software-Defined Networking is a new emerging technology that is quickly gaining popularity amongst the largest corporations. However, this new networking paradigm has a centralized point of failure at the controller. With this choke point, it is imperative that it be designed with security at the forefront. This research aims to shed light on one of the possible ways that having a centralized point of failure in the network can provide malicious attackers an avenue to disrupt an entire enterprise network. Two experiments are performed. The first experiment confirms a vulnerability in a hardware switch discovered during this research. The second, to see if generating fake malicious software switches on the network is enough to inflate the Java Virtual Machine Heap to capacity and cause the controller to crash.

These tests are conducted using enterprise grade servers and Software-Defined Networking enabled hardware switches. All trials conducted to confirm the discovered vulnerability resulted in a 100% success rate and revealed the cause of the vulnerability is due to the switch not following standards set forth by the Open Networking Foundation. The experiment performed to test the possibility of inflating the JVM Heap to capacity resulted in a 0% success rate. However, prolonged intermittent Denial-of-Service is achieved by causing the Garbage Collector to stop performing Young Generation Garbage Collections and to perform Full Garbage Collections exclusively. This transition to exclusive Full Garbage Collections is a result of the attack inflating the heap memory allocated to the Floodlight controller.

The system reaches a point where the only way the Garbage Collector can keep the heap from running out of memory is by performing Full Garbage Collections. Moreover, the execution of Full Garbage Collections grew in both frequency and

duration. The time spent per Full Garbage Collection began exceeding one second, compared to thousandths of a second during normal operation. The mean total time the JVM spent Garbage Collecting was 2053.987 seconds. During these collections the controller is halted in its execution which is the root cause as to why the attack was unable to reach the level of utilization required to throw the *OutOfMemoryError*, as well as the cause of the prolonged intermittent Denial-of-Service.

Acknowledgements

I am thankful to the Air Force for providing me the opportunity to attend AFIT and continue my technical education. I am especially thankful to my advisor Dr. Barry E. Mullins and committee members Dr. Timothy H. Lacey and Dr. Michael R. Grimaila for their efforts to ensure the quality of this work. Most importantly, I am thankful to my family, who have provided support and guidance from the other side of the country.

Anthony A. Portante

Table of Contents

Page

Page

Page

List of Figures

List of Tables

List of Abbreviations

ANALYSIS OF DENIAL-OF-SERVICE ATTACK VECTORS IN SOFTWARE-DEFINED NETWORKS

I. Introduction

This research implements and analyzes a network attack methodology that can be used to attack the controller of a Software-Defined Network (SDN) using a tool developed by the author that masquerades as a physical SDN switch. This chapter introduces and provides context to the problem, the approach used to address it, the research goals, and expected results. Lastly, the assumptions and limitations for this research are explained, followed by how this research contributes to the field of network security.

1.1 Background

As computer networks evolve in complexity and size the need for network administrators to be able to view their enterprise network(s) from the top down has grown. Along with obtaining a global view of their network, administrators have also pressed industry to move towards a single point of control allowing them to modify and configure the network as they need from a single host. After the release of OpenFlow in 2007 [8], Software-Defined Networking was able to provide administrators with these capabilities by separating the data plane and control plane of a tradition network, thus allowing for dumb switches and a centralized controller. The controller contains all the intelligence needed to make networking decisions, rather than traditional networks, which have the intelligence built into each switch.

These SDN controllers also possess the capability to allow the network administrator to manually add routes and direct traffic at will from the controller by using OpenFlow, which allows switching devices on the network to be configured via a controller. Controllers can run custom software developed either by vendors or by a network administrator. This custom software can provide other features to the network, such as Quality of Service and security. These features allow an administrator to exert control over a network from a centralized point using software, compared to traditional networks which require manual configuration.

1.2 Problem Statement

SDN is a new emerging technology that is quickly gaining popularity amongst the largest corporations. However, this new networking paradigm has a centralized point of failure at the controller. With this choke point, it is imperative that it be designed with security at the forefront. This research develops, implements, and analyzes a Denial-of-Service (DoS) attack vector that can only be leveraged due to the introduction of SDN. This research studies what can occur if an attacker compromises a host on an SDN with the objective of causing a DoS from within the network by executing an "out-of-the-box" attack strategy. Ultimately, this research provides insight into some of the attack vectors a malicious user can utilize in order to bring down an SDN.

1.3 Research Goals and Hypothesis

Two open-source controllers currently on the market are the Floodlight and OpenDaylight Controllers. Both controllers have industry support from major vendors such as Google, Cisco, HP, Big Switch Networks, Pica8, etc., and both controllers are written in Java [9] [10]. This research focuses on the Floodlight controller and aims to

take advantage of the fact that since it is programmed in Java, it runs in the Java Virtual Machine (JVM), which has very limited memory growth potential. The objective of this research is to inject malicious software switches into the network with the goal of causing the controller to artificially inflate in memory, thus consuming more heap memory than the JVM is allocated. This inflation would in turn, cause the JVM to throw an *OutOfMemoryError* making the controller program crash. This research hypothesizes, if enough malicious switches are injected into the network, the Floodlight controller will artificially inflate the utilized JVM Heap memory, in turn, consuming the allocated memory for the JVM Heap and triggering the *OutOfMemoryError*. By ensuring the generated malicious switches survive Garbage Collections, the attack can build upon itself, consuming more memory as more malicious switches are continuously generated.

1.4 Approach

A network consisting of two hosts (virtual machines), hosted on a SuperMicro SuperServer and a physical Pica8 P-3290 SDN switch are deployed. One host is running the Floodlight controller software and the other is representing the compromised host. The compromised host executes a script that generates the malicious switches that connect to the controller, while the controller host runs the Floodlight controller software. JVM Heap memory, Garbage Collection statistics, and OS memory are monitored on the controller host continuously throughout the experiment. These measurements are used to confirm the hypothesis and to confirm the success or failure of the attack.

1.5 Assumptions

As with any experiment, there are several assumptions that must be made. When it comes to performing an attack over a network, many of these assumptions must always be true for the attack to work. The following assumptions are used for this experiment:

1. The controller is Floodlight Version 1.2
2. The JVM settings on the controller's host are left to default settings for JVM-64 bit
3. Transport Layer Security (TLS) is not in use by the controller
4. The compromised host can communicate to hosts on the controller's subnet
5. The Pica8 switch is fully dedicated to the experiment with no other hosts or network segments other than what is required for the experiment

1.6 Contributions

This thesis contributes to the security of SDN as a whole by examining how OpenFlow and the controllers introduce a DoS attack vector that previously did not exist in traditional networks. It is easier for developers to design secure software and hardware when the attack vectors are known. Traditional DoS attacks are well known and studied; as such, this research exposes an "out-of-the-box" DoS attack vector that developers have not previously been required to defend against. This exposure allows the SDN developers to design security features to defend against such attacks.

1.7 Thesis Overview

This thesis is organized into five chapters. Chapter 2 defines important security concepts, SDN, the JVM, as well as presents relevant research in the realm of attacks performed on SDN's. The methodology for evaluating the attack as well as the approach to the problem are discussed in Chapter 3. Chapter 4 presents the results of the experiment and the analysis of the collected metrics. Lastly, Chapter 5 summarizes the research and concludes with an explanation of the significance of this research and suggested future work.

II. Background and Related Research

This chapter defines the current industry standard in regards to security principles and discusses some of the traditional terminology used in both security and network attack. Section 2.1 introduces the standards of providing computer and network security. The concept of network attack is discussed in Section 2.2 along with important examples that pertain to this research. A thorough review of Software Defined Networking is discussed in Section 2.3 by introducing the data and control planes, SDN controllers such as the Floodlight Controller, and the OpenFlow Protocol. Section 2.4 introduces the Java Virtual Machine and an in depth discussion of the heap memory model and Garbage Collector. In Section 2.5, security concerns that are caused by the introduction of SDN to networks is discussed and examples of such concerns are provided. Lastly, Section 2.6 concludes with work related to this body of research.

2.1 Security Principles

Enterprise security is a complicated and complex task that concerns all devices within the enterprise from the end hosts to intermediate switches and routers. Traditional enterprise security is comprised of network security and host security [11]. The goal of network security is to provide protection for the enterprise network from unauthorized access. This is generally done via some form of network examination of the data traversing the network in order to detect intrusions into the network and the connected computers. Additionally, there are many tools that can be implemented to help provide more layers of security within the network, such as Intrusion Detection Systems, encrypted control channels, and firewalls. Host security is performed by creating and enforcing specific policy models that include using access control, user permissions, software patches, and specified authentication schemes [11]. This

research does not fall within the scope of host security; however, it is important to note that host and network security often are not mutually exclusive and do rely on each other to provide a holistic security architecture for enterprise networks. The network class is the main area of focus in this research and as such, it is important to discuss the industry standard model of a secure network. The following attributes are the aforementioned standards [12] [13]:

- Access - only those who are authorized can communicate on the network
- Confidentiality - information is protected while in transit
- Authentication - users and hardware are verified to be who they claim to be
- Integrity - if data is modified in transit, then the change can be detected
- Non-repudiation - each user or system can be held responsible for actions perpetrated on the network

These attributes are what provide the foundation for network administrators to architect a secure network. This means that everything from the security software to the network monitoring and administration tools, all need to play a part in securing the network. SDN is one such tool that is growing at an ever-increasing rate; however, as with all new computing and networking technology there are going to be software flaws that leave the system vulnerable to exploitation and attack.

2.2 Network Attack

The concept of network attack deals with what an attacker can accomplish by taking advantage of the network architecture or vulnerability found in either the network-capable software or hardware. This process is depicted in Figure 1. The

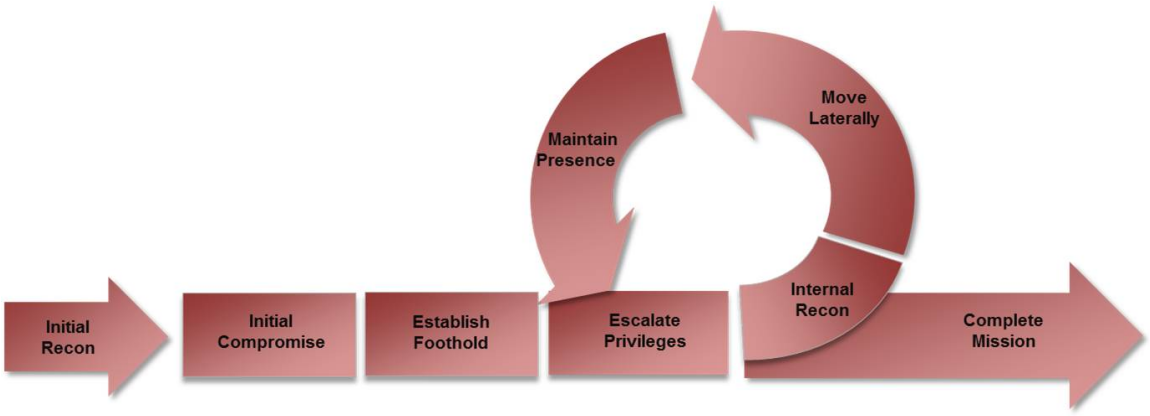

Figure 1. Mandiant Attack Lifecycle Model Showing Steps Taken During a Cyber Attack [1]

attacker gains access to the network by compromising a host within the network or by authenticating into the network using stolen credentials.

Included in Mandiant's APT1 report [1], Figure 1 illustrates a common attack methodology called the "Attack Lifecycle Model", which details the steps taken during a cyber attack. The network attack actions take place throughout the "Attack Lifecycle"; however, this research operates under the assumption that the attacker has already achieved Step 3 and has established a foothold on a host within an SDN. Once on the network, some more common network attacks are the following [14]:

- Eavesdropping (Sniffing) - an attacker who gains access to the network can listen in and intercept traffic over the network
- Data Modification - modify captured data in transit
- Identity Spoofing - using a fake network identity to communicate, such as a fake IP or MAC address
- Password-Based Attacks - network sniffing may turn up usernames and passwords being transmitted over the network, which can be used to log into systems

- Denial-of-Service - preventing normal use of networks and networked resources, generally via network flooding, causing the network infrastructure to crash or discontinue service

- Application Layer Attack - consists of deliberately attacking applications running on top of either the servers or hosts within the network

This research consists of an attack that is both a DoS and Application Layer Attack. This combination is due to the goal of causing the controller to crash, which in turn causes the network to cease operations (DoS), and since the controller is an application running on a Linux host on the network, this research also produces a Application Layer Attack.

2.2.1 Vulnerability.

In order for an attacker to perform a successful network attack, the attacker must take advantage of a vulnerability that lies within the network. A vulnerability is a flaw within a system that allows it to be exploited and then used for malicious purposes [11]. This vulnerability can take shape in many forms such a flaw in software code, hardware, system configurations, security architectures, or even a human user. Some of the most serious flaws are known as Zero-day vulnerabilities. A Zero-day is a vulnerability that is not publicly known and there is not a patch currently available. Figure 2 depicts the number of Zero-day vulnerabilities reported by Symantec over the years depicting a bullish trend over the last five years.

Software Defined Networks are new in the computing world and, with such infancy, are prone to containing Zero-day vulnerabilities and require extensive security testing such as testing the system against attack methodologies that do not exist in traditional networks. Results from this research help contribute to the comprehensive testing required to ensure that SDN is ready for real world exposure.

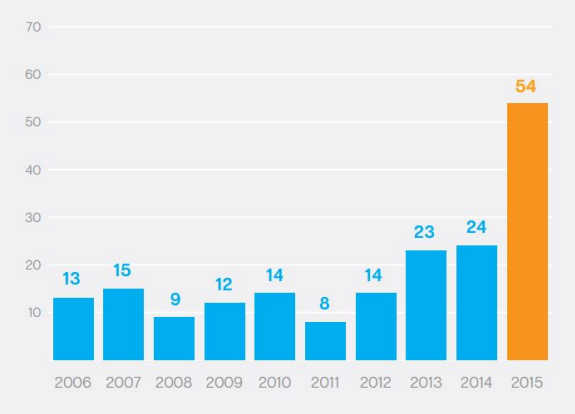

Figure 2. Symantec Security Report 2016: Zero Days Reported Annually Displaying a Bullish Trend [2]

2.2.2 Exploit.

The term "exploit" is defined as taking advantage of a specific vulnerability that exists either in code or in specific systems architecture. This action can range from exploiting a flaw in a program's code, taking advantage of a networking protocol, or even exploiting flaws in the architecture of a network built by the administrator. The most prized goal of any exploit is to gain control over the asset being exploited. However, this level of control may not always be the end goal of an attacker. For example, a nation-state attacker may wish to exploit a vulnerability found on the enemy network architecture by causing the network to go down along with all communication abilities [15].

2.2.3 Insider Threat.

An insider threat is one of the most detrimental computer security threats a system administrator faces. It occurs when a person or persons have some form of special knowledge or access to sensitive systems or information [16]. This insider is on the network and generally is already past most external defenses, with legitimate privileges and access to internal systems. The two most common types of insider

threat are the intentional and the unintentional. The intentional threat is most often a disgruntled employee who actively plans to attack the network from within. The unintentional insider is someone with access and unintentionally grants an outside entity access by falling victim to a social engineering attack. A deep examination of insider threats is outside the scope of this research; however, it is important to note these threats since this research is performed with the assumption that the attacker has already compromised a host on the internal network.

2.3 Software Defined Networking

Software Defined Networking is a new networking architecture technology that decouples the network control plane and the data (forwarding) plane that are currently intermingled in traditional networking technology. This new paradigm takes strict control of transmission and other forwarding functions away from networking devices, such as network switches and grants a software controller the authority to dictate flow. This allows the data and control planes to be on separate lanes of traffic and reduces the amount of intelligence that needs to be built into switches, thus severely cutting costs. This section explains in detail the composition of Software Defined Networking.

2.3.1 Switches and Routers.

Switches and routers are networking devices whose job are to forward network packets from one port and/or interface to another with the goal of doing so as close to line speed as possible. Due to the fact that speed is so imperative, these devices are managed by a network administrator and do not inherently keep any records (logs) of communications beyond hardware and network addresses [17]. This lack of record keeping is an intentional design choice since keeping detailed records is computationally expensive and would require enterprise grade devices to be unreal-

istically expensive. These hardware devices generally process traffic in microseconds and are also not updated based on the changing state of the end hosts, with the exception of customized solutions [18]. In order to achieve dynamic control of the switch configuration, SDN removes the control aspect that has to be built into traditional switches, which usually consists of hundreds of thousands of lines of code. This level of intelligent control that has to be built into enterprise grade switches made them very expensive. By allowing a software controller to handle decision making, SDN switches can be less costly and directly programmable. This research specifically uses the Pica8 P-3290 SDN switch.

2.3.2 Control and Data Planes.

In traditional networking systems, the control plane is the medium in which the network administrator exerts control over the network by configuring and sending commands to the network devices. The data plane is the medium in which the network traffic traverses to and from the network. In these traditional networks, these two planes must share the same communication lanes and in turn, consume resources and leave the network in a rigid state. The main goal of SDN is to create a networking system that could turn a rigid environment (traditional network systems) it into a flexible system. As described in Casado et al. the ideal control plane is one that is flexible, simple, and vendor neutral [19]. The authors also emphasize the importance of having a data plane that includes intelligent devices having virtual switching features built into them. The current state of networking technology does not meet any of these goals and in an effort to realize the aforementioned goals, requires the current networking device vendors to surrender proprietary control and simply transfer the data. SDN gives the management control of the traffic over to a software controller. Figure 3 displays how the SDN architecture separates the data

(infrastructure) and control planes - (i) physical hardware and (ii) the software that exerts control over the hardware respectively [3]. The application layer is a component of the controller software that provides developers access to an Application Program Interface (API) to allow them to interact with the control layer.

The Northbound API lies between the control layer and application-layer and allows developers to affect the control plane. is referred to as the Northbound API. This API enables the developer or user (network administrator) to perform calls to the control layer, on which the SDN controller is located. The controller provides the management capabilities formerly owned by the network infrastructure and thus performs the routing functions for the network. The infrastructure layer represents the data plane and consists of physical or virtual hardware that follows the commands sent forth by the controller to perform routing functionalities. It is between the control layer and infrastructure layer that these planes interface and the OpenFlow Protocol (OFP) is used for communication.

2.3.3 OpenFlow Protocol.

The OFP is the bridge that allows the infrastructure to communicate with the controller. OpenFlow is an application layer protocol that allows the controller to communicate with the switches on the network in areas such as routing particular traffic, creating, or deleting new routing rules, etc. The Open Networking Foundation (ONF) is a non-profit foundation tasked with the development and accelerating the adoption of SDN [20]. ONF has built a massive member base of over 100 companies with over 64 OpenFlow products on the market. The most current OFP release at the time of this research is version 1.5. This version, however, is still not widely used or supported by most hardware; as such, version 1.3 is used in this research due to the amount hardware support.

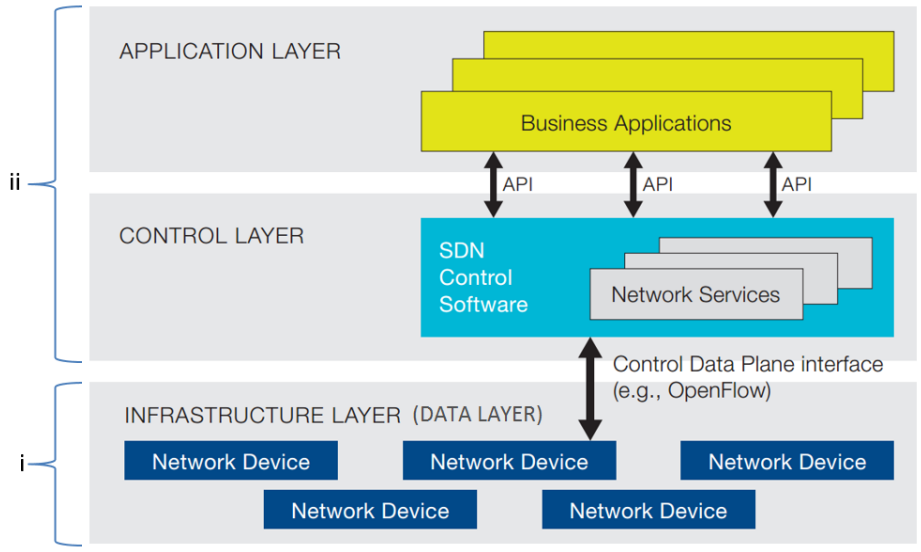

Figure 3. Open Networking Foundation: SDN Architecture Displaying Separation of Data and Control Planes [3]

2.3.4 Floodlight SDN Controller.

The Floodlight controller is an open-source controller sponsored by Big Switch Networks. In this research, the Floodlight SDN controller (version 1.2) is used due to its overwhelming popularity within the SDN community and the fact it is open-source. The Floodlight controller is written in Java and is well integrated into the Eclipse IDE, which provides easy access to any developer [4]. Built using the Apache Ant framework, Floodlight has exploded in popularity and has become even more accessible to programmers, thus fostering a very active community of developers [21]. Figure 4 gives a glimpse into the Floodlight framework as well as the included modules built in. The Representational State Transfer (REST) API is the user-facing API (Northbound API) that allows the system or network administrator to query the controller or send commands.

The Module Application section is the area of the controller the switches on the network talk to using the OFP. It is through the Module Application section new switches connect and continue communication with the controller. This section then

uses a Java API to communicate with other key pieces of the Floodlight controller. This communication is an important piece, as it shows all switches are communicating with a section of the controller that calls upon a Java API in order to process packets, create objects, store data in memory, etc. - all of which are crucial to the success of the hypothesized attack.

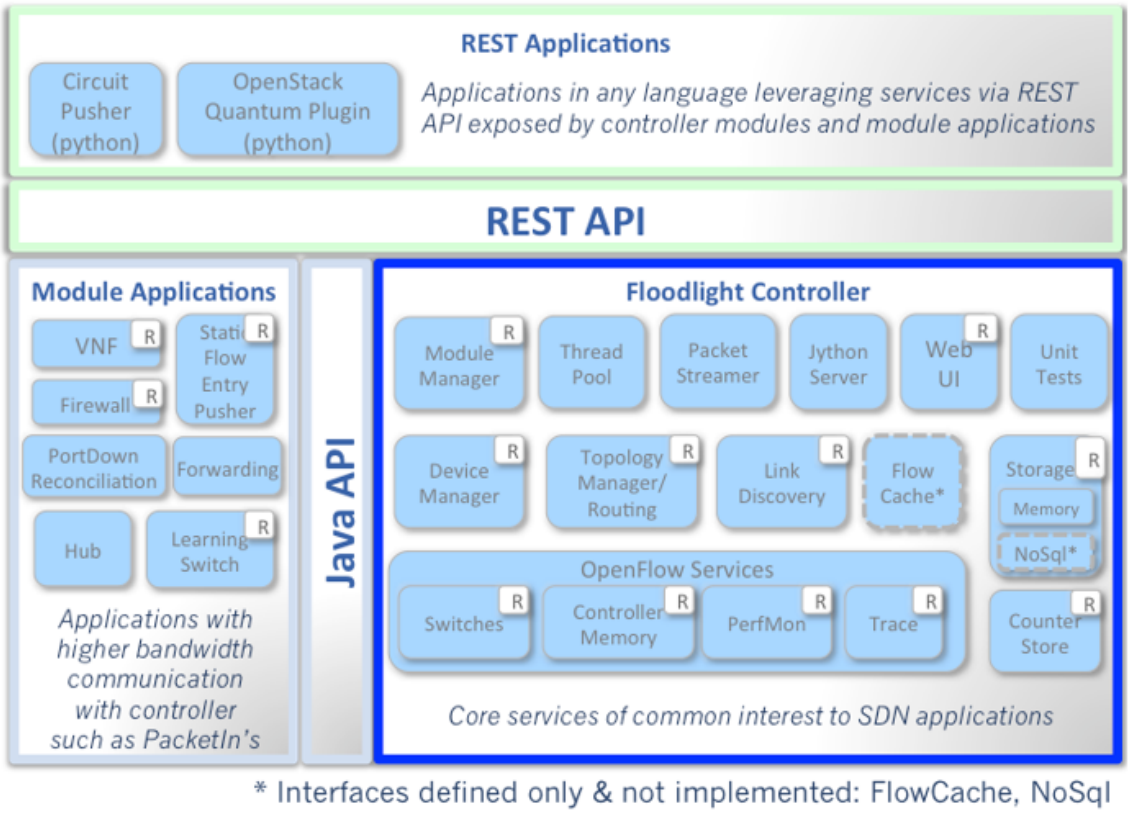

Figure 4. Project Floodlight: Floodlight Framework Included Modules [4]

2.3.5 Controller/Switch Handshake.

In order for a SDN OpenFlow enabled switch to associate with the controller, the controller and switch must complete an OpenFlow handshake. Shown in Figure 5, the handshake process begins by completing a TCP 3-way handshake. The controller is constantly listening on TCP port 6653 (default port) and waits for the switch to

initiate the connection. Upon the successful establishment of a TCP connection the controller sends an OFPT_HELLO message. This message tells the switch the highest OFP version it supports. The switch then replies back to the controller with its own OFPT_HELLO message telling the controller the highest OFP version it can support. The switch and controller then compare what the other is able to support and selects a version compatible for both parties. This negotiation means if the controller says it can support version 1.4 and the switch says it can support version 1.3, then version 1.3 is the highest version support by both and thus will be the protocol version used for all future communications.

After the agreed protocol version is established, the controller sends a OFPT_FEATURES_REQUEST message to the switch. This message is used by the controller in order to identify the switch MAC address and to learn its basic capabilities, such as the number of tables and maximum number packets it can buffer. Lastly, the switch responds with a OFPT_FEATURES_REPLY message containing the requested information. At this point the OpenFlow handshake is complete and OpenFlow messages can now be freely exchanged [5] [6].

2.4 Java Virtual Machine

The Floodlight Controller is written in the Java programming language and as such, the code must be executed within the JVM. The JVM is an abstract computing machine responsible for Java's hardware and operating system independence. Similar to a real (physical) computing machine, the JVM has its own instruction set and manipulates various sections of memory during run time [22]. Figure 6 shows the layout of the JVM memory model as well as other components that allow the JVM to operate properly. This research focuses heavily on the JVM Heap and Garbage Collector. The remaining pieces of the JVM are outside the scope of this research

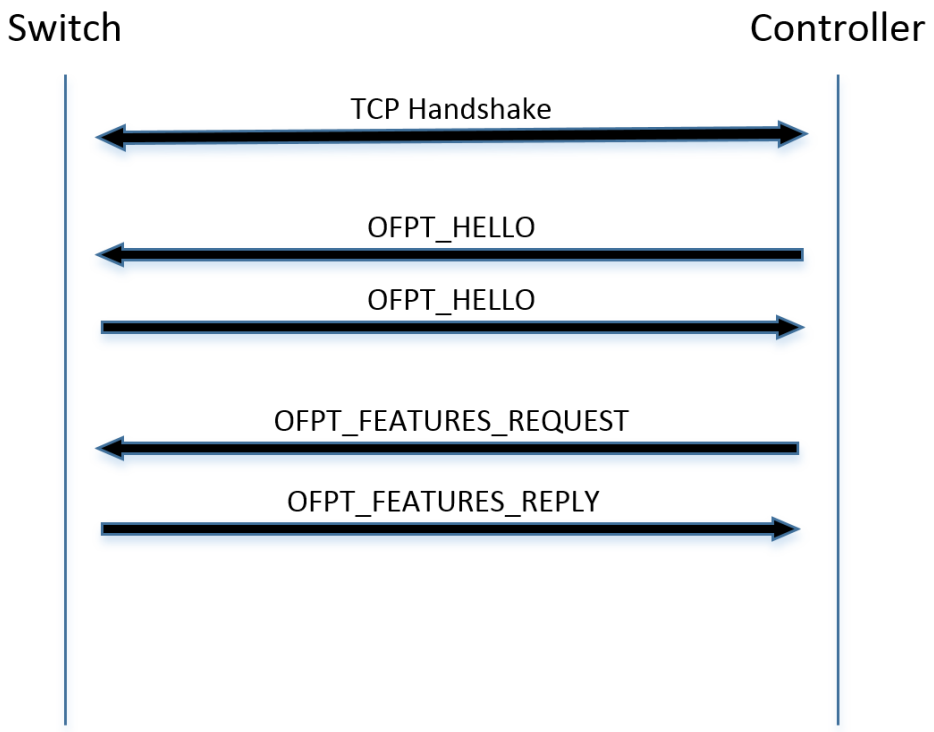

Figure 5. Controller/Switch Handshake [5] [6]

and are not discussed further; however, it is important to note these pieces still exist for when analysis of the memory is discussed in Chapter 4 with regards to the OS memory.

2.4.1 Garbage Collector.

The Garbage Collector is in charge of freeing up space in the heap in order to allow for the allocation of new objects. This is performed by reclaiming the memory of objects that are no longer referenced. Objects that still are being referenced are left alone and their age or tenure is incremented by one. The Garbage Collector then compacts the objects remaining so no gaps remain between them. There are three type of Garbage Collections: Young Generation Garbage Collection (minor), Old Generation Garbage Collection (major) and Full Garbage Collection (full). The

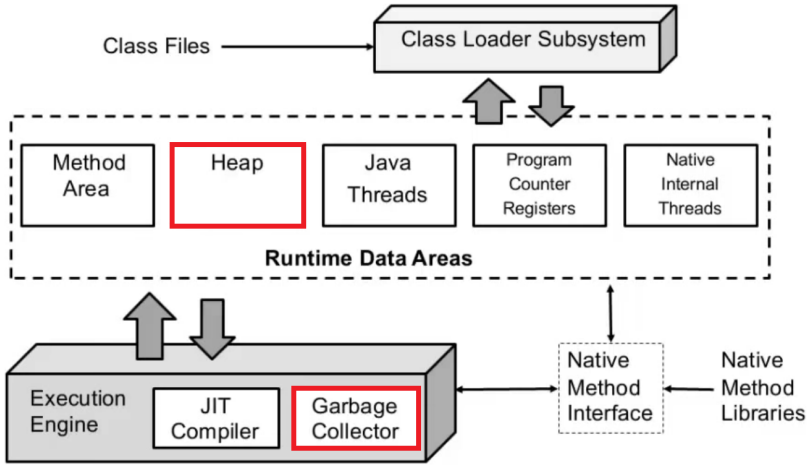

Figure 6. Java Virtual Machine: Memory Model [7]

Young and Old Garbage Collections are performed in the Young and Old heap spaces respectively, while the Full Garbage Collection consists of a collection on the entire heap. This removal of unused objects ensures memory is being used in the most efficient manner possible.

The garbage collection process causes a pause in the program's execution called a Stop-The-World (STW) event, which is when all threads of execution are gracefully paused in order to allow the Garbage Collector to perform its job [7]. The JVM offers multiple garbage collection algorithms a user can choose; however, this research is performed under the assumption that JVM defaults are left intact. Not all of Garbage Collection algorithms trigger a STW event; however, the vast majority do, including the default algorithm ParallelGC, or Parallel Garbage Collector [23]. The Garbage Collection process is elaborated in greater detail in Chapter 3.

2.4.2 Java Virtual Machine Heap.

The JVM Heap is designed using a Generational Memory Model where the heap as a whole is divided into separate "generations" each of which contain only certain data (objects) based on the age or tenure of the data. Shown in Figure 7, the heap

space is divided into 3 generations: the Young Generation (YG), the Old Generation (OG), and the Permanent Generation (PG). The YG is designated for new or recently created objects and is divided even further into three spaces; Eden, Survivor Space 0 (S0), and Survivor Space 1 (S1), these spaces are discussed in greater detail in Chapter 3. Garbage Collections in the YG are frequent, fast (e.g., hundredths of a second), and efficient due to the fact that objects in the YG are likely to be short lived [7].

The OG or Tenured space is for older objects that have survived multiple Garbage Collection events in the YG, therefore the JVM "promotes" the object and moves it into the OG Space. The OG is typically larger in memory capacity than the YG and grows at a slower rate dependent on the program being executed. Garbage Collection events in the OG are infrequent but take significantly longer to complete ranging from a tenth of a second to multiple seconds, again, dependent on the program being executed and the amount of memory needing to be parsed. The PG stores meta data and Java classes or types of data that should remain in the heap permanently in order for proper program execution.

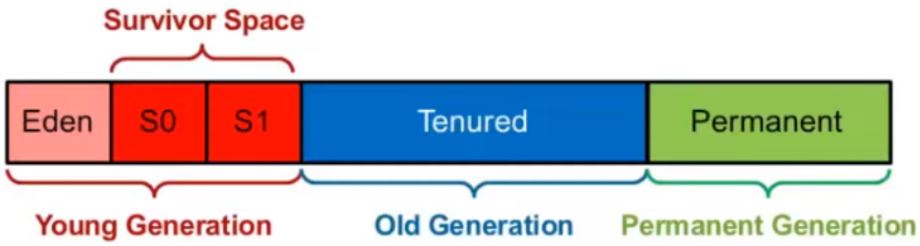

Figure 7. Java Virtual Machine: Heap Memory Model [7]

2.4.3 Dynamic Heap Sizing.

It is important to note, each of these spaces (Eden, S0, S1, OG, PG) have their own memory capacity assigned to them by the JVM. These capacities can grow or shrink

in size by the JVM based on a percentage of free space within a specific generation. The default *MinHeapFreeRatio* and *MaxHeapFreeRatio* for 64-bit Ubuntu 14.04.05 using Java 8 is 0% and 100% respectively. These parameters mean if the percent of free space in a generation falls to 0%, then the generation is expanded. This can continue up to the maximum allowed size of the generation. Similarly, if the free space reaches 100%, then the generation is contracted, again subject to the minimum size of the generation [24]. Separating the heap into generational spaces allows the JVM to perform specified memory management only to the areas that require attention providing a very efficient and powerful memory model.

2.5 SDN Security Concerns

Security is at the forefront of today's computing and networking systems. Vulnerability analysis on traditional networking infrastructure has provided network administrators and industry alike the knowledge to build their networks in a more secure fashion and create hardware and software tools to further secure traditional networks. That being said, traditional networks are by no means 100% secure; however, they are more mature and the security vulnerabilities are well known and understood. SDN on the other hand is a new networking technology still maturing and is not nearly as well known and understood as traditional networks.

SDN also introduces new components to the networking paradigm such as the OpenFlow Protocol and the controller. These new components are young and immature relative to traditional networking infrastructure and protocols. These new components could possibly be used as new network attack surfaces. While SDN has the ability to use TLS, the rapidly evolving nature of SDN has caused vendors of both the switches and the controllers to either not fully implement TLS, not implement it at all, or include an implementation that simply is infeasible to use in an enterprise

network. The infeasibility is due to the amount of overhead required by the administrator to configure it properly. Thus, TLS is nearly never used in SDN enterprise networks [25]. Since TLS is essentially not implemented, this leaves the network open to Man-in-the-Middle attacks between the switch and controller. Another security concern is the fact there is absolutely no authentication between the switch and the controller [26]. This lack of authentication provides an attacker the ability to associate with either the switch or controller in order to launch attacks on either party. The attacker can also spoof messages to either the switch or controller and possibly force the switch to route traffic to the attacker, allowing the attacker to sniff network traffic not intended for them on a switched network.

Another important security concern of SDN is its vulnerability to traditional DoS attacks. Since switches can only store a limited amount of flows in their flow table, it is possible for attackers to send bogus traffic on the network to cause their tables to reach capacity. At this point, all new *Packet-in* messages need to be forwarded to the controller, causing the control plane to become congested and slow down dramatically [27]. The last security concern is since controllers are software, they also run into the limitations and security flaws of the programming language in which they are written. A controller written in C/C++ may be susceptible to buffer overflow and memory vulnerabilities. A controller written in Java must be executed within the JVM, which severely restricts the amount of memory the controller has access to, providing the perfect storm for the controller to scale too much and run out of memory. If these vulnerabilities are exploited to the proper extent, it is highly likely the controller will immediately crash, and without a controller, the network will no longer be able to provide service.

2.6 Related Research

Much of the work relating to SDN research is performed using Mininet in order to emulate a SDN network or is a security survey of the high level functionalities of SDN [27]. This research uses physical hardware such as a Pica8 P-3290 SDN enabled switch and enterprise grade servers.

Kevin Benton et al. conducted an OpenFlow Protocol vulnerability analysis in which the authors discussed a high level security review of the protocol, concluding many of the vulnerabilities discussed in Section 2.5, such as switch authentication and DoS risks [26].

Gregory Pickett gave a presentation at the DefCon 22 Security Conference in August 2014, where he discussed tools he built that would exploit the OFP version 1.0 including an XML injection vulnerability on the OpenDaylight Controller that would give the attacker root access once completed. He also built a tool that performs a DoS flood causing the network to be brought offline [28].

III. Methodology

3.1 Problem Statement and Goals

This research focuses on the Floodlight Controller and aims to take advantage of the fact it is programmed in Java and thus in order to run, must be executed in the JVM. The JVM has a very limited memory growth potential and places all objects created by the running process into the dedicated heap space within JVM as described in Chapter 2. Everything is represented as an object in Java, and thus the heap space gets used frequently. According to Oracle documentation, the initial heap size is 15.6% of the machine's physical memory and the initial maximum heap size is 25% of the machine's physical memory [29].

The objective of this research is to inject malicious software switches into the network from a compromised host with the goal of causing the controller to artificially inflate in memory and attempt to allocate more heap memory than JVM is allocated. This in turn causes JVM to throw an *OutOfMemoryError* making the controller program crash [30]. If the controller fails and is offline, the SDN it is in charge of ceases functioning, as no new flows can be created. This experiment aims to discover if SDN perpetuates new attack vectors not present in traditional networks.

3.2 Vulnerability Discovery

Preliminary experimentation was performed in order to calibrate the optimal test parameters and workloads for repeated testing. During these preliminary tests it was discovered the Pica8 P-3290 SDN switch does not follow the ONF switch specifications for OFP version 1.4 when its flow tables are full [5] [31]. Although the attack tool used in this research uses OFP version 1.3, the Pica8 uses version 1.4 when connected to Floodlight since 1.4 is the highest protocol version supported by both

the Pica8 and Floodlight. This results in triggering an unforeseen DoS vulnerability in the switch, creating a limiting factor that cannot be circumvented and altered how the experimentation was executed going forward. The limiting factor is that the vulnerability is getting triggered at premature point in the trial, preventing the experiment from inflating the JVM Heap memory to the needed levels.

As such, two experiments were created with the first experiment being a repeatability test in order to verify the vulnerability in the switch and the second, being the memory attack on the controller. Since the vulnerability was found via preliminary tests on attacking the controller using the attack tool, that is discussed in Section 3.3.1, the same attack technique and tool is used in both experiments. More extensive details of the switch vulnerability is discussed in Chapter 4.

3.3 Approach

In order to verify the discovered vulnerability in the Pica8 P-3290 switch, an environment with a host, SDN switch, controller, and attack tool are required. Figure 8 shows the experiment setup. The Floodlight controller (version 1.2) and the compromised host are Linux virtual machines running on an ESXi server. These hosts are then connected to the Pica8 P-3290 switch with all network traffic flowing through the Pica8 switch.

In order to test if the attack on the controller is successful, an environment with a host, controller, and attack tool are required. Figure 9 shows the experiment setup. The Floodlight controller (version 1.2) and the compromised host are Linux virtual machines running on an ESXi Server. Due to the discovered vulnerability, the Pica8 switch is removed from the experiment setup. Instead, both the controller host and the compromised host are connected to the same physical Network Interface Card (NIC). The ESXi server recognizes this configuration and creates a virtual switch.

This setup is analogous to setting up two virtual machines on a home desktop. If virtual machine A wishes to send packets to virtual machine B, the NIC simply forwards the packets to virtual machine B, without having to first send the packets to a router. In essence, this setup allows the compromised host to connect directly to the controller, bypassing the limitation introduced by the Pica8 P-3290 switch.

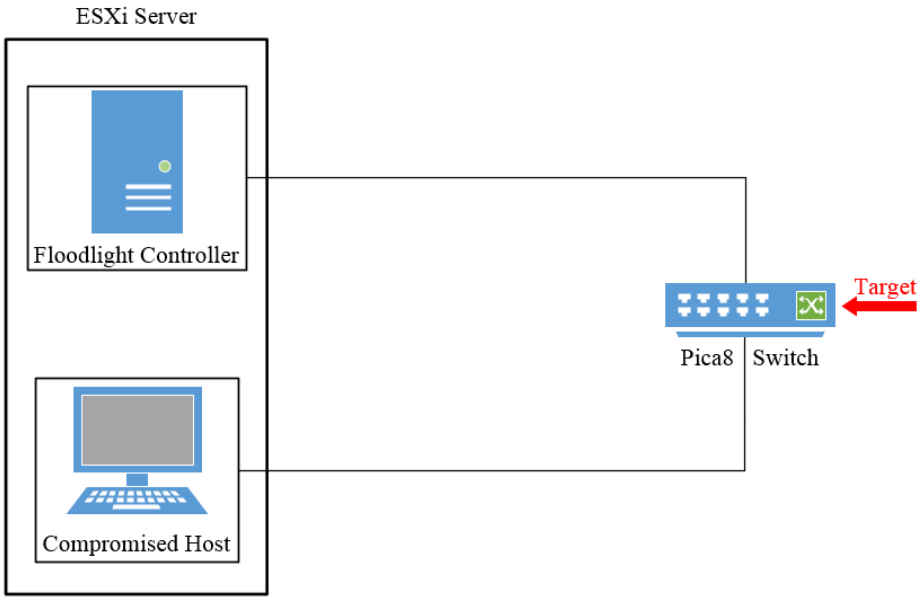

Figure 8. Experiment 1: Switch Attack Experiment Configuration

3.3.1 Malicious Software Switch.

The Malicious Software Switch (MSS) is the attack tool used in both experiments that connects to the controller in order to launch the attack. This tool is written in C and is capable of connecting to the controller, completing the OpenFlow Switch/ Controller handshake as specified by the ONF switch specifications for OFP version 1.3 [5]. Upon completion of the handshake, the MSS is now registered by the controller (Floodlight version 1.2) as a valid switch on the network and is able to send OpenFlow messages to the controller at will. This tool is also capable of answering any queries

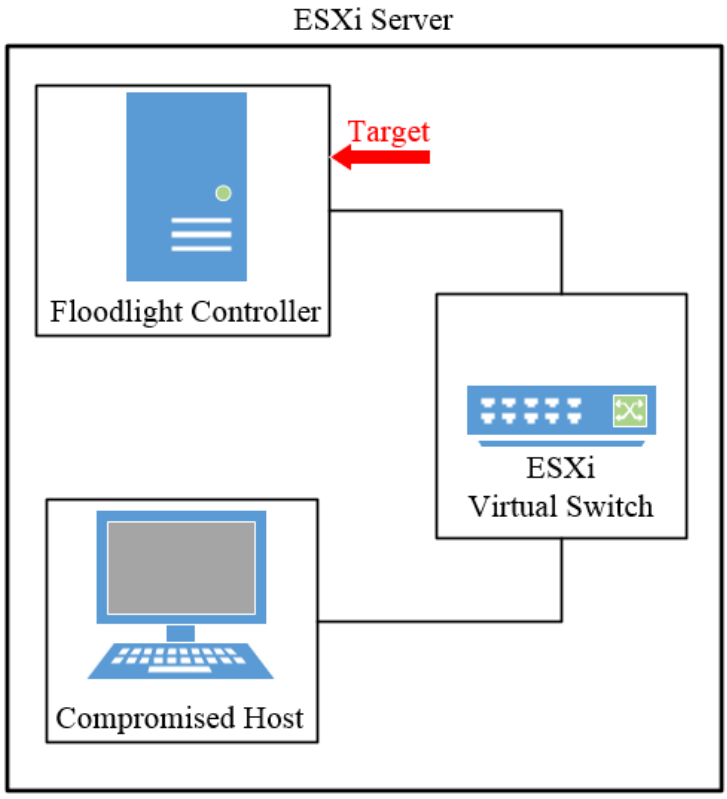

Figure 9. Experiment 2: Controller Memory Attack Experiment Configuration

sent by the controller with valid responses that would be expected from a physical hardware switch, which is required in order to maintain connection.

In order for the MSS to pass as a real switch on the network, it needs to populate the fields in its OFP responses to the controller with information that appears legitimate. This data is collected by analyzing Wireshark captures of the Pica8 P-3290 SDN switch as it associates with the Floodlight controller. This data is then extracted and coded into the MSS with random number functions used to generate MAC addresses. When the MSS is registered by Floodlight, it appears as if it is a real Pica8 switch on the network.

3.3.2 Artificial Memory Inflation.

When a new object is created by Floodlight, such as a new switch connecting to it, this object is allocated in the Eden space within the YG. This process of allocating new objects in the Eden space continues until the Eden space utilization has reached capacity. As shown in Figure 10, when the Eden space reaches capacity, a Garbage Collection event is triggered and those objects that are still being referenced have their age incremented (illustrated as a value within the object rectangle) and moved to one of the survivor spaces, either S0 or S1. Shown in Figure 11, after the migration of objects is complete, the Eden space, as well as the opposing survivor space, is cleared. For example, if a Garbage Collection is called and there are referenced objects in the Eden space and S1 space, those referenced objects are incremented in age then moved to the S0 space. Then the Garbage Collector clears the Eden and S1 space of all remaining unreferenced objects.

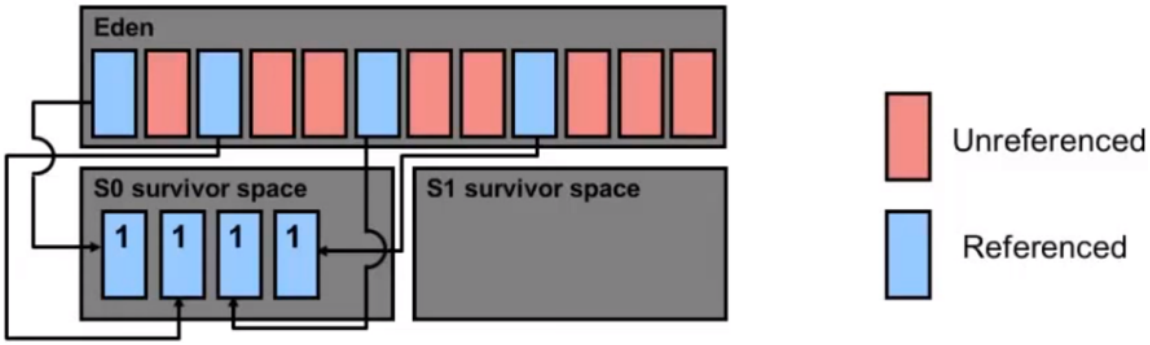

Figure 10. Garbage Collection: Eden Space [7]

As illustrated in Figure 12, once this process is finished and the Eden space fills up again, the same Garbage Collection process occurs, except now the JVM looks at objects that are still referenced in both the Eden space as well as the currently occupied survivor space. Those objects still referenced have their age incremented and then are moved to the unused survivor space. Lastly the Eden space and the

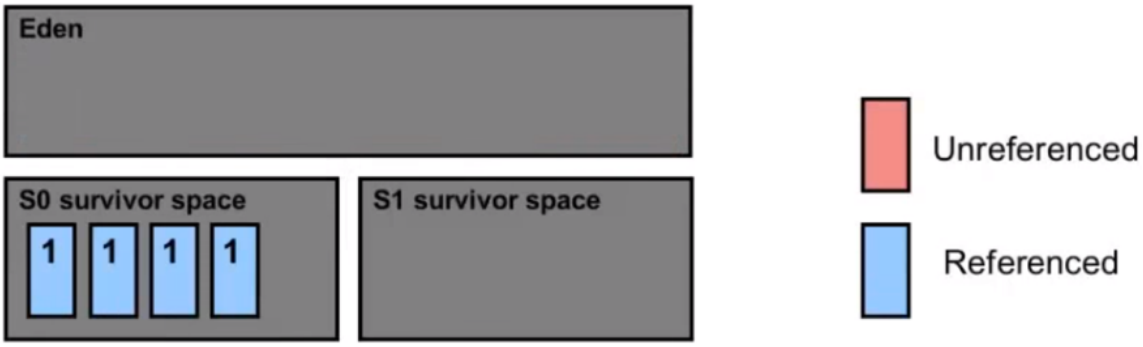

Figure 11. Eden Space post Garbage Collection [7]

previously used survivor space are cleared again. Shown in Figure 13, this process repeats until an age threshold is met, at which point, objects that have reached this threshold will be promoted from the YG to the OG, where the value of 9 is an example age threshold for object memory promotion.

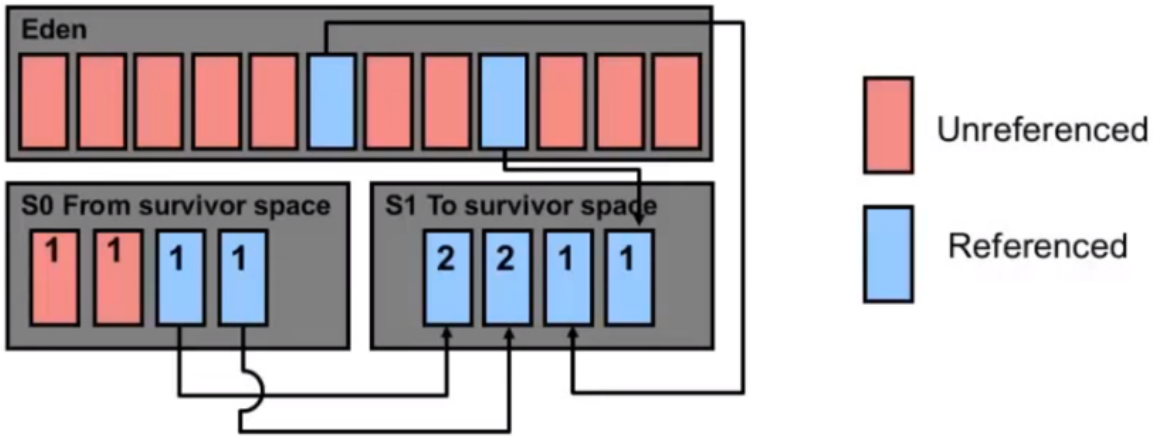

Figure 12. Garbage Collection: Survivor Space Migration [7]

Throughout the execution of a Java program such as the Floodlight controller, if the capacities of each section of memory are continually met, the JVM dynamically increases their respective capacities. This also happens if there is a such large quantity of objects that survive the Garbage Collection that the designated survivor space or Tenured Spaces capacity is not sufficient to store them all. Garbage Collection events

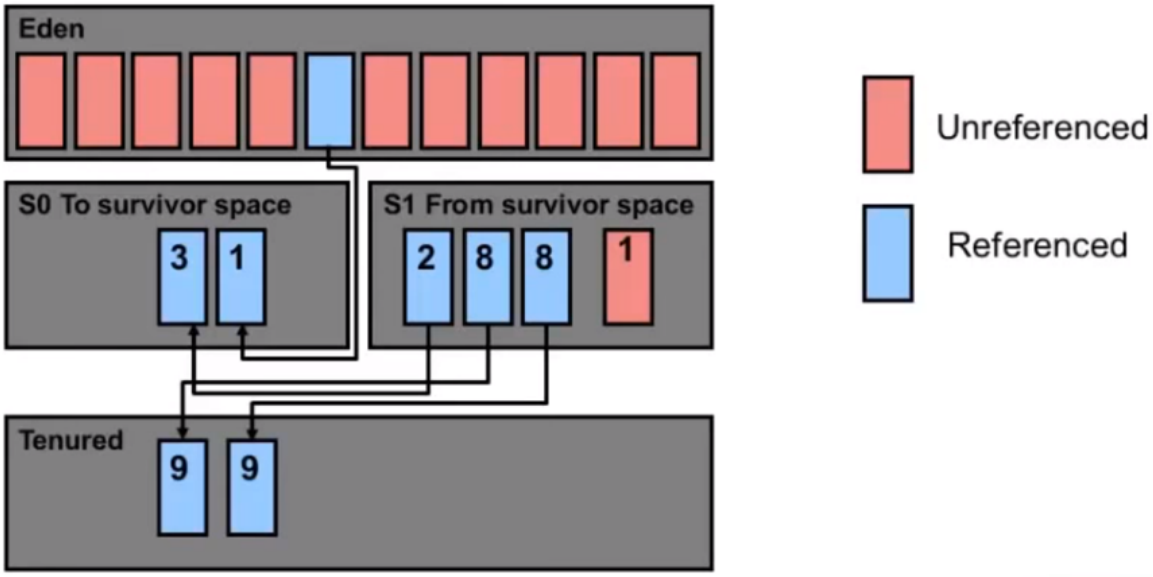

Figure 13. JVM Object Promotion: Young Generation to Old Generation [7]

in the OG are rare relative to the YG and also consume significantly more time and resources. As such, Oracle recommends avoiding Old Generation Garbage Collections as much as possible [7]. Since the MSSs are not physical switches on the network, if they can survive Garbage Collections they will cause an artificial increase in the capacities and utilization of the Young and Old Generation heap spaces to the point where the JVM reaches its maximum heap size, running out of memory and throwing the desired, from the attackers perspective, *OutOfMemoryError*.

3.4 System Boundaries

The System Under Test (SUT) is the switch attack and controller memory attack process and is shown in Figure 14. The components of the system are the Floodlight SDN controller (version 1.2), Pica8 P-3290 SDN switch, and the compromised host (MSS). These Components Under Test (CUT) produce metrics for evaluating the process. The workload going into the system is the number of connected MSSs. It is important to distinguish that the Pica8 P-3290 SDN switch is not used in the con-

troller memory attack experiment and is only used in the switch attack experiment. All other components of the SUT are present in both experiments.

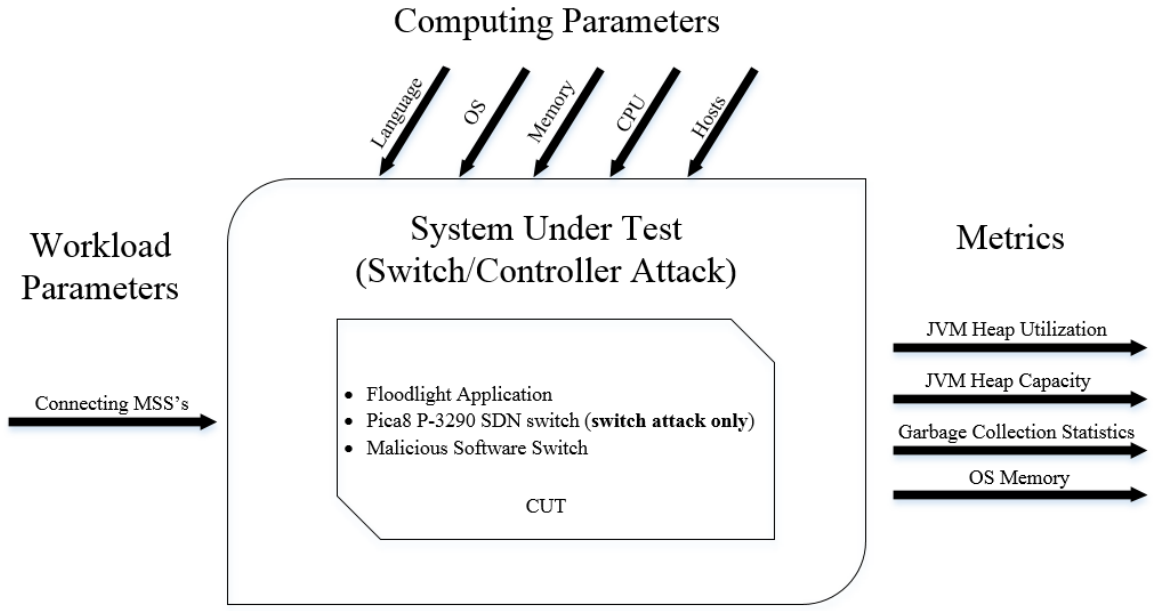

Figure 14. System Under Test with Components Under Test

3.5 Parameters and Factors

The switch attack and controller memory attack experiments both use workload and system parameters where *factors* are the parameters that vary during the experiments. The remainder of this section defines and explains the parameters and how each one relates to the SUT. There are parameters that are held constant such as the programming language, OS, memory, CPU, hardware switch, and number of hosts. Table 1 presents the factors, factor levels, and the amount of time data collection is performed for each experiment.

Table 1. Factors

Factor	Experiment	Factor Level	Experiment Duration
Malicious Software Switches	Switch	2,000	1,900 seconds
Malicious Software Switches	Controller	4,000	4,400 seconds

3.5.1 Factors.

As mentioned in Section 3.2, since the vulnerability on the switch was discovered via preliminary tests on attacking the controller using the MSS, the same attack technique and tool is used in both experiments. This is why the factor for both experiments is the same in Table 1.

3.5.2 Factor Levels.

The Factor Level column of Table 1, represents the number of MSSs that will be generated for each experiment. For the switch attack experiment, 2,000 MSSs are generated. The preliminary tests that discovered the vulnerability, generated 2,000 MSSs. Although the highest observed value (during preliminary tests) of MSSs needed to connect to the controller in order to trigger the vulnerability in the switch was 1,454, the quantity of 2,000 was kept in order to maintain the same factor level used during the initial discovery. This also provides a buffer in case the range of MSSs need to trigger the vulnerability have a larger range than expected (e.g., 1,454-1,800).

For the controller memory attack experiment, 4,000 MSSs are generated. Preliminary tests were conducted in order to calibrate the factor level needed to assure the attack on the controller would raise the JVM Heap memory usage to the levels needed to throw the *OutOfMemoryError*. These tests reveled that 4,000 MSSs was an adequate factor level, thus is the quantity used for the controller memory attack.

3.5.3 Experiment Duration.

The Experiment Duration column in Table 1 represents the length of time in seconds each experiment is executed. This duration also represents the amount of time the metrics for each experiment are collected.

The duration for the switch experiment is set to 1,900 seconds. This is because during preliminary testing the highest observed value of MSSs needed to connect to the controller in order to trigger the vulnerability in the switch was 1,454. Since the compromised host spawns 1 MSS/second, the best case is after 1,454 seconds, 1,454 MSSs would be connected. In order to account for some MSSs losing connection or delayed connection due to the sheer volume of connections to the controller, the value of 1,454 was rounded up to 1,500 and 400 seconds were added summing to 1900 seconds. The quantity of 400 is added because during preliminary tests, it was observed that terminating the experiment and the metric collections as soon as the vulnerability is triggered does not provide enough data to show the results of triggering the vulnerability. Further tests showed that adding 400 seconds provides an adequate amount of time to observe the results of the vulnerability being triggered.

The same concept is therefore applied to the controller memory attack. Since the attack consists of 4,000 MSSs being generated, an extra 400 seconds was added, bringing the duration of metric collection to 4,400 seconds for the controller attack experiment.

3.5.4 Workload Parameters.

3.5.4.1 Connecting Malicious Software Switches.

As seen in Figure 14, the workload parameters are the connecting MSSs. Each test consists of the compromised host spawning a large number of MSSs. Each MSS created is its own process and each is created via the same executable with the only

varying property being each MSS randomly generates a MAC address. The number of MSSs the compromised host spawns is in the Factor Levels column of Table 14.

3.5.5 System Parameters.

Figure 14 also presents the computing parameters as language, OS, memory, CPU, hardware switch, and hosts. The attack tool used (MSS) is comprised of multiple .c and .h files compiled into an executable using gcc (Ubuntu 4.8.4-2ubuntu1 14.04.03) 4.8.4. This version is retrieved by running the command *gcc –version* on the Ubuntu host representing the attacker. The compromised host is allocated 8 GB of RAM and 4 cores (8 logical), and the controller is allocated 8 GB of RAM and 8 cores (16 logical). These allocations allow for the attack to be unhindered due to system resources. All traffic for the switch attack experiment is routed through the Pica8 P-3290 SDN switch and all traffic for the controller memory attack experiment is routed through ESXi virtual switching as discussed in Section 3.3.

3.5.6 Metrics.

There are 4 different metrics collected in order to analyze the attack process. The JVM Heap Utilization metric measures the amount of heap being used by the controller in kilobytes. The utilization of each space in the heap (Eden, S0, S1, Tenure, Permanent) is tracked and later combined to provide an analysis of the utilization of the heap as a whole. This metric allows insight into the fluctuations of memory throughout the attack process as well as verify the attack is causing the memory to increase.

The JVM Heap Capacity metric measures the capacity of each heap space in kilobytes. Similarly, these individual measurements are later combined to provide a holistic view of the memory capacity growth throughout the attack process. By

measuring the heap capacity growth, it allows for the verification that the attack is capable of forcing the controller to artificially inflate to its maximum allowed size in memory.

Garbage Collection Statistics are the last of the JVM metrics collected. These statistics consist of the number of minor collections, number of full collections, and the total amount of time the Garbage Collector spent running during the duration of the experiments. These statistics are important to analyze due to the fact a Garbage Collection is a Stop-the-World (STW) event, meaning the total amount of time the collector was spent running is also the amount of time the controller spent *not* running. This data also provides a glimpse into how the JVM is attempting to handle the attack by use of its Garbage Collector.

Lastly, the OS Memory metric measures the amount of memory the Floodlight Controller is occupying in RAM in kilobytes. This metric only sees the memory used by the entire running application and not the individual pieces that make up the application. This perspective of the OS memory means all parts of the JVM will influence this metric, not just the heap space.

3.6 Experimental Setup

3.6.1 Overview of Setup.

The test environment includes one physical server and one hardware switch. The server is a SuperMicro SuperServer X9QR7-TF+/X9QRi-F+ with two Xeon E5-4610 v2 processors, eight 1000BASE-T network interface cards, 16 cores, and 512 GB of RAM. The switch is a Pica8 P-3290 SDN Switch [32] with OpenFlow 1.2-1.4 support. The SuperMicro server hosts two virtual machines for both experiments. The compromised host is an Ubuntu 14.04.05 (4.4.0-45-generic) host and has 8 GB of RAM and 4 physical cores (8 logical). The controller host is also an Ubuntu 14.04.05

(4.4.0-45-generic) host and has 8 GB of memory 8 cores (16 logical). The Switch attack has all traffic routed through the Pica8 switch, while the controller attack has all traffic routed through an ESXi virtual switch.

3.6.2 Assumptions.

These assumptions are a reiteration of those that are found in Section 1.5, but are necessary to mention again as both experiments will not execute properly unless the following assumptions are met:

1. The controller is Floodlight Version 1.2
2. The JVM settings on the controller's host are left to default settings for JVM-64 bit
3. TLS is not in use by the controller
4. The compromised host can communicate to hosts on the controller's subnet
5. The Pica8 switch is fully dedicated to the experiment with no other hosts or network segments other than what is required for the experiment

If the compromised host on the network does not have a way of accessing the network the controller's host is on, there is no way for the attack to be executed, as it requires connection to the controller. Moreover, the Pica8 switch needs to be fully dedicated to the experiment as to not skew results. If there are other hosts running on the network while the attack is taking place, these hosts could end up consuming flows on the switch, which adds unnecessary variables to the experiment.

3.6.3 Experiment 1: Switch Attack.

Preliminary tests showed that after approximately 1,454 MSSs connected to the controller through the Pica8 switch, the switch's flow tables became filled. However,

instead of following the ONF switch specifications [5] [31], the switch sends out *TCP Zero-Window* packets (receive window is set to 0) to the controller, while continuing to send OFP messages to the controller. This causes timeouts to occur on both the controller's side and the attacker's side resulting in MSS connections being dropped by the controller. This also creates a complete DoS as now the controller is no longer able to create new flows. Due to MSSs being disconnected, the attack cannot persist since the quantity of MSSs that are needed for a successful test cannot be reached.

This was an unknown flaw in the Pica8 switch and thus a new test was performed to verify that the vulnerability can be repeatedly triggered, as well as to see if it is also detectable using the metrics described in Section 3.5.6. Figure 15 provides a high-level view of the attack prior to its execution and after the attackers script has completed. The script generates a new MSS every second, this timing was selected as preliminary experiments showed that the controller was unable to handle large amounts of simultaneous switch connection attempts.

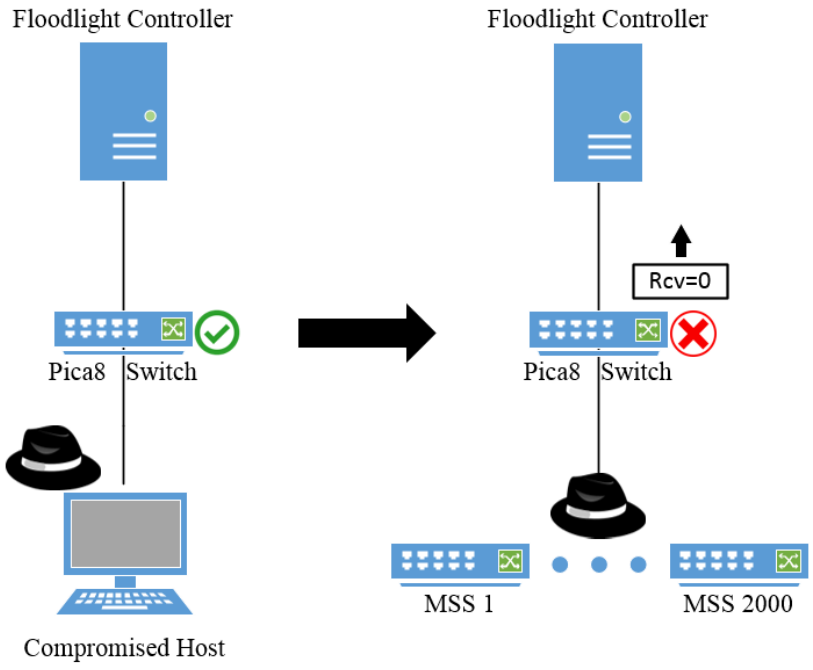

Figure 15. Experiment 1: Switch Attack Before and After

3.6.4 Experiment 2: Controller Memory Attack.

The experiment consists of a compromised host executing a script that creates multiple separate copies of the MSS. When an OpenFlow switch successfully connects to the controller and completes the OpenFlow handshake, the controller allocates a new object for that switch. Moreover, since the controller is tasked with keeping track of all devices on the network, it will also allocate more memory to keep track of each device characteristics and capabilities. This experiment investigates if creating enough MSSs on the network can cause Floodlight to artificially inflate and run out of heap memory. Figure 16 illustrates the experiment prior to the execution of the attack and then shows the experiment after the script has completed, providing a visual of how the second experiment unfolds. As mentioned in Section 3.6.3, the script generates a new MSS every second.

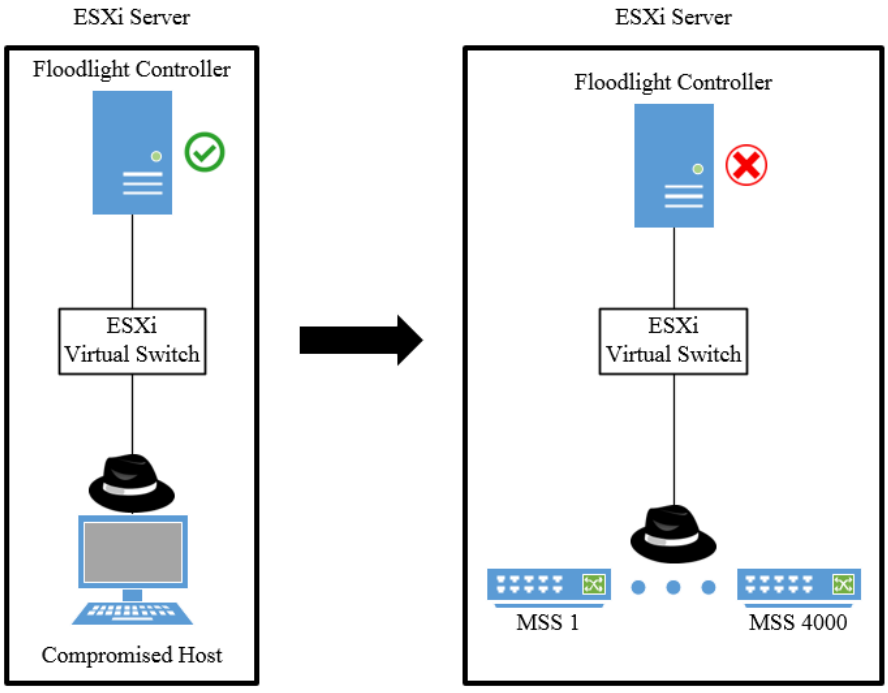

Figure 16. Experiment 2: Controller Memory Attack Before and After

3.6.5 Experimental Scripts.

In order for the experiment to run, multiple scripts are executed in order to start Floodlight, gather metrics, and execute the compiled C programmed attack tool (MSS.c).

3.6.5.1 RunExperiment.sh.

RunExperiment.sh is executed on the controller host and is the first script run in both experiments. This script shown in Figure 17 begins the Floodlight Application and redirects all output to *Controller_log.txt*, then writes the starting time to a text file on the desktop (*/home/controller2/Desktop/Times.txt*).

```
1 #!/bin/bash
2 ‘java -jar target/floodlight.jar >> Controller_log.txt‘
3 date >> /home/controller2/Desktop/Times.txt
```

Figure 17. RunExperiment.sh Script

3.6.5.2 JVM_mem_monitor.sh.

JVM_mem_monitor.sh is executed on the controller host after RunExperiment.sh and is shown in Figure 18. This script first appends the starting time to the same text file containing start times that *RunExperiment.sh* wrote to (*/home/controller2/Desktop/Times.txt*). The script then finds the Process ID (PID) for the Floodlight application and stores the value into a variable, seen on line 5. Lines 7-10 output the current JVM memory settings such as the initial heap size and max heap size. Lastly line 12 runs the command (*jstat -gc $PID 1000 1900*) which gathers the JVM Heap Capacity, Utilization, and Garbage Collection statistics, where *$PID* is the variable containing the Floodlight Application PID, *1000* is the amount of time

in milliseconds between data collections, and *1900* is the number of times data needs to be collected. The value *1900* is for the switch attack experiment and is replaced with *4400* when used to monitor the controller memory attack experiment. When executed, this script has all output redirected to a text file (.txt) with a description and number designator signifying which experiment and the trial number (e.g. *bash ./JVM_mem_monitor.sh >> SwitchAttack_JVM_5.txt*).

```
1  #!/bin/bash
2
3  date >> /home/controller2/Desktop/Times.txt
4
5  PID=‘ps -ef | grep ’[j]ava’ | awk ’{print $2}’ ‘
6
7  echo ""
8  echo "JAVA VIRTUAL MACHINE MEMORY INIT (bytes):"
9  echo ""
10 java -XX:+PrintFlagsFinal -version | grep -iE
   ’HeapSize|PermSize|ThreadStackSize’
11
12 jstat -gc $PID 1000 1900
```

Figure 18. JVM_mem_monitor.sh Script

3.6.5.3 OS_mem_monitor.sh.

OS_mem_monitor.sh is executed on the controller host after JVM_mem_monitor.sh and is shown in Figure 19. This script first records the start time to same text file containing start times that *RunExperiment.sh* and *JVM_mem_monitor.sh* wrote to (*/home/controller2/Desktop/Times.txt*). The script then outputs only the column titles for the *ps aux* command, shown on line 5. Lastly, lines 7-11 loop for the number of seconds the experiment is run. The value *1900* is used for the switch attack experiment, while *4400* is used for the controller memory attack experiment.

The body of the loop contains the command that gathers the amount of memory in RAM the Floodlight Application is occupying (RSS). After this data is gathered, the script pauses for 1 second (*sleep 1s*), then resumes execution. This pause is so both *JVM_mem_monitor.sh* and *OS_mem_monitor.sh* gather data in the same intervals. When executed, this script requires root access and has all output redirected to a text file (.txt) with a description and number designator signifying which experiment and the trial number (e.g. *sudo bash ./OS_mem_monitor.sh >> SwitchAttack_OS_5.txt*).

```
1  #!/bin/bash
2
3  date >> /home/controller2/Desktop/Times.txt
4
5  ps aux | head -n1
6
7  for i in {1..1900}
8  do
9         ps aux --sort -rss | sed -n '2p'
10        sleep 1s
11 done
```

Figure 19. OS_mem_monitor.sh Script

3.6.5.4 AttackScript.sh.

AttackScript.sh is executed on the compromised host and is the last script executed. First the script will loop for the number of MSSs that need to be generated (2,000 for the switch attack/4,400 for the controller memory attack) and is seen in lines 5-9. The body of the loop contains the command that runs the MSS executable (*'./file 10.231.1.200 6653 -n' &*), where *file* is the executable, *10.231.1.200* is the IP address of the controller, *6653* is the TCP port to connect to, and *-n* tells the program to maintain connection to the controller indefinitely. This command then puts

the newly created process in the background to keep CPU utilization to a minimum. The last command in the loop (*sleep 1s*) pauses the script for 1 second, due to the controller being unable to handle large amounts of simultaneous switch connection attempts, as discussed in Section 3.6.3.

Since the switch attack experiment duration is 1,900 seconds, line 13 is commented out as no delay to allow the experiment to complete is needed after the 2,000 MSSs are generated. For the controller memory attack experiment, line 13 is uncommented and the script will pause for 430 seconds (*sleep 430s*). The value 430 is used because the controller memory attack experiment is executed for 4,400 seconds, so after the 4,000 MSSs are generated the experiment will need to continue for another 400 seconds. An extra 30 seconds were added to provide a safety buffer to make sure the attack script is the last script to stop executing.

In lines 15-22, the script begins to tear down all the generated MSSs. This is performed to allow for ease of repeated testing. Line 16 gather all of the PIDs for the generated MSSs and then loops through the PIDs. The body of the loop then outputs that the script is going to kill the process, kills the process, then pauses for 1 second. This pause is needed because if the script tries to kill all the processes without any delay in between, errors where thrown by the terminal. When executed, this script outputs all error messages to */dev/null* because during experimentation MSSs may throw socket errors when the controller terminates the connection (e.g. *bash ./AttackScriptr.sh 2> /dev/null*).

```
#!/bin/bash

echo "Spawning switches...please wait"

for i in {1..2000}
do
        `./file 10.231.1.200 6653 -n` &
        sleep 1s
done

echo "I have finished!"

#sleep 430s

echo "Beginning tear down process...please wait"
for j in `ps -aux | grep "./file" | head -n2000 | awk {'print $2'}`
do
        echo "Killing PID: $j"
        kill -9 $j 2> /dev/null
        sleep 1s
done
echo "Tear down complete"
```

Figure 20. AttackScript.sh Script

3.6.6 Test process.

Both experiments consist of a set of trails. In each trial, the following steps are followed:

1. Execute *RunExperiment.sh* on the controller host
2. Execute *JVM_mem_monitor.sh* on the controller host
3. Execute *OS_mem_monitor.sh* on the controller host and type in root password
4. Execute *AttackScript.sh* on the compromised host
5. (SWITCH ATTACK ONLY) Begin Wireshark on the controller host in order to capture the *TCP Zero-Window* packets

The Wireshark capture is started last for two reasons. The first is by making it last, the first four steps of the test process remain identical across both experiments, preventing possible bias in the order of data recording. The second number of MSSs that need to connect to trigger the vulnerability in the switch is 1,454 MSSs (as mentioned in Section 3.6.3) and the compromised host is only spawning a MSS once per second, at the minimum 1,454 seconds would need to pass before any expectation of capturing the *TCP Zero-Window* packets.

3.6.7 Floodlight User Interface Monitoring.

During execution of the trials for both experiments, progress is monitored using the Floodlight web UI shown in Figure 21, which is accessed on the controller host web browser by browsing to: *http://127.0.0.1:8080/ui/pages/index.html*. Red boxes are drawn on Figure 21 to highlight the three key areas being monitored. Area 1 displays the number of switches connected and associated with the controller. This is purely a numerical value and updates in real time once a switch (MSS or Pica8) has finished the OFP handshake (Section 2.3.5) and allows for continuous monitoring of the ongoing trial. Area 2 is a dial showing the JVM Heap Utilization as well as where the utilization is in relation to the JVM Heap Capacity. This dial moves and updates in real time; however, since Garbage Collections can remove large amounts of data in fractions of a second, the dial can move to seemingly arbitrary values. This indicator is used to have "situational awareness" as to what is occurring regarding the Garbage Collector/Heap memory. Lastly, area 3 is in essence the same as area 2, but the data is provided numerically rather than visually. This area is again used to have a big-picture understanding of what is occurring with the Garbage Collector/Heap memory. This UI provides a great deal of information during the switch attack and controller attack trials allowing for continuous monitoring as the attacks unfold in

real time, which provides more knowledge as to what exactly is occurring.

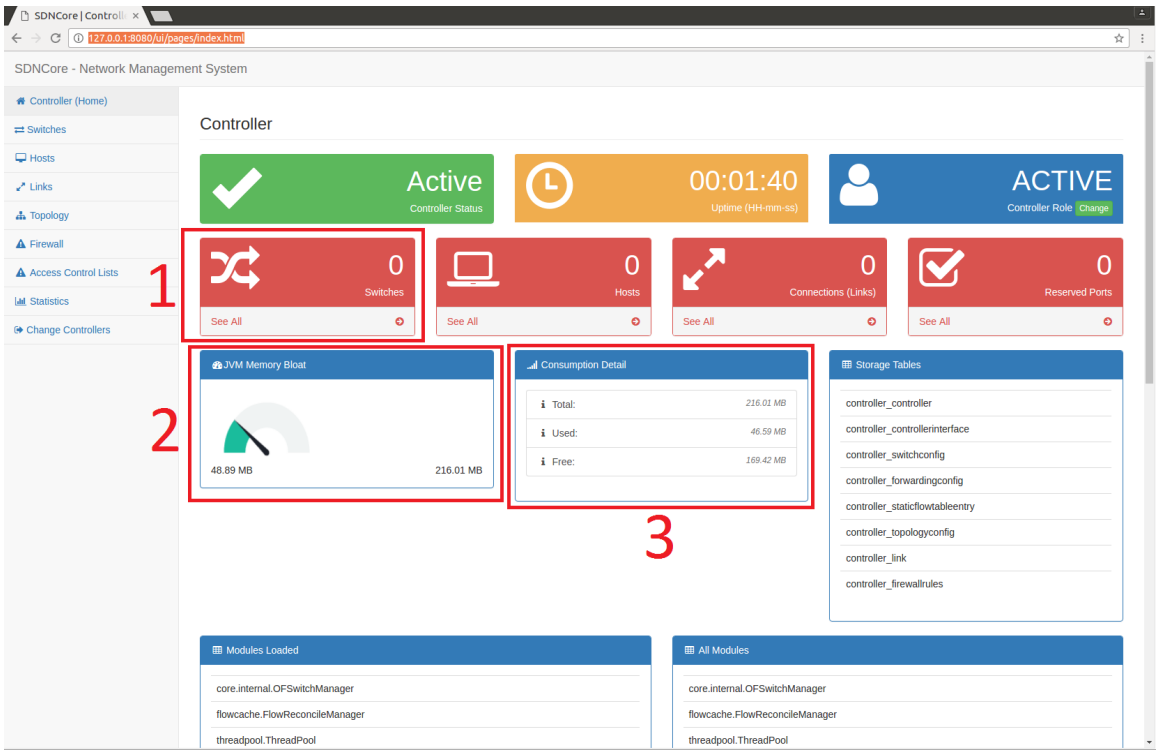

Figure 21. Floodlight UI Displaying Memory Utilization

3.7 Experimental Design

3.7.1 Overview.

Two experiments are designed to address the goal discussed in the beginning of this chapter as well as to address the shortcoming of the Pica8 switch discussed in Section 3.2. In order to minimize experimental bias, all virtual machines are hosted on identical systems. Additionally, the attack host and the controllers host used for both experiments maintain the same memory and CPU allocations discussed in Section 3.5.5. Each trial is executed in the identical order across the experiments. The switch attack experiment is executed for 10 trials and the controller memory attack experiment is executed for 30 trials. Only 10 trials were needed for the switch

attack experiment due to the mean data being consistently inside the 95% confidence interval and further experimentation was no longer required. Similarly, 30 trials are executed for the controller attack experiment as after 30 trials the due to the mean data being consistently inside the 95% confidence interval and further experimentation was no longer required. Further details on the results and data consistency of the experiments is discussed in Chapter 4.

3.7.2 Data Processing.

Figure 22 depicts a high-level overview of the process used to evaluate the captured data. For each run, the raw data is recorded in a text file format (.txt). All JVM data (utilization, capacity, Garbage Collections statistics) is collected by the same script and recorded to the same file, while the OS memory is gathered by another script and recorded on a separate file. Next these files are converted from text files to Excel files (.xlsx) in order to be more easily handled by R. From within R, the data for each metric from each trial is then pulled and put into a single Excel file for each metric for each experiment (controller or switch). For example, the JVM Utilization, Capacity, and Garbage Collection statistics as well as the OS memory for all 30 trials of the controller experiment are placed into 4 separate files respectively, each with 30 columns representing the respective 30 trials. This is also performed for the switch experiment. Once the data is consolidated, the mean and 95% confidence interval is calculated for each metric. Lastly, the means and confidence intervals are plotted using R and exported for analysis.

3.7.3 Test Timing.

These two experiments are not an evaluation of timing performance and as such, precise time synchronization was not required nor does it impact the results of the

experiment. All scripts and executables are run in the order discussed in Section 3.6.6 and take approximately 7-10 seconds to execute.

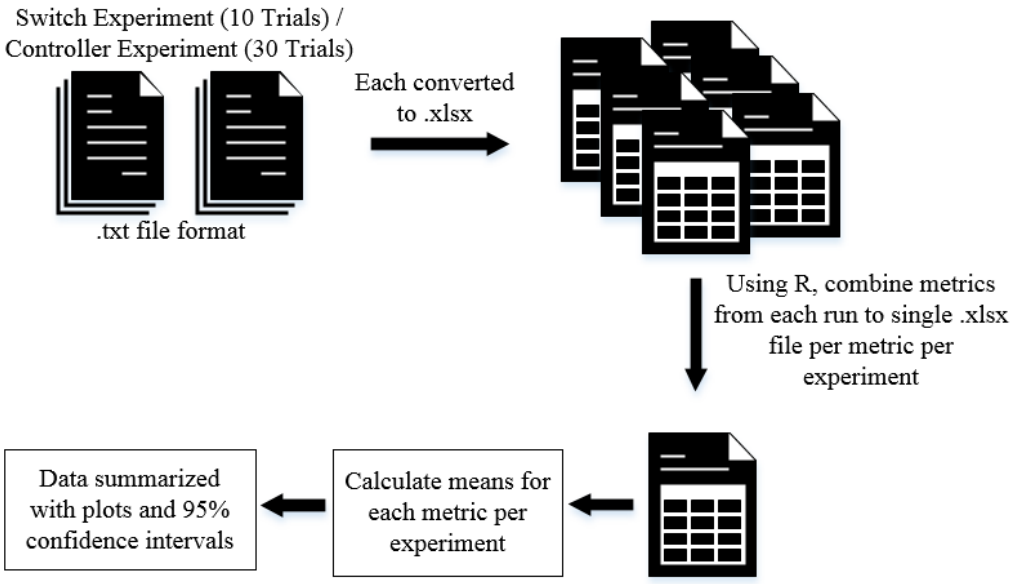

Figure 22. Data Processing Diagram

3.8 Evaluation Technique

In order to evaluate the results from the metrics gathered, R is used to perform the statistical analysis. In total, 4 metrics are analyzed using R - JVM Heap Utilization, JVM Heap Capacity, JVM Garbage Collection statistics, and OS Memory.

The JVM Heap Utilization is comprised of

$$JHU = Eu + S0u + S1u + Ou + Pu \tag{1}$$

where Eu is the Eden space utilization, S0u is the S0 space utilization, S1u is the S1 space utilization, Ou is the Old Generation space utilization, and Pu is the Permanent Generation space utilization.

The JVM Heap Capacity is comprised of

$$JHC = Ec + S0c + S1c + Oc + Pc \tag{2}$$

where Ec is the Eden space capacity, S0c is the S0 space capacity, S1c is the S1 space capacity, Oc is the Old Generation space capacity, and Pc is the Permanent Generation space capacity.

The JVM Garbage Collection statistics are actually multiple sets of independent data consisting of number of YG Garbage Collections, number of Full Garbage Collections, cumulative time spent on YG Garbage Collections, cumulative time spent on Full Garbage Collections, and total cumulative time spent on all Garbage Collections. These metrics are all returned in finished form and do not require a summation or modification for analysis.

Similarly, the OS memory is the Resident Set Size (RSS) returned by a terminal command as seen in Section 3.6.5.3. The RSS is the amount of memory a process consumes in RAM in kilobytes. This metric also does not require summation or modification for proper analysis.

3.8.1 Scatter Plot Smoothing.

As mentioned, the 95% confidence interval is calculated for each metric. These intervals are created for each data point, meaning it is performed for 1,900 data points for the switch attack and performed for 4,400 data points for the controller memory attack. However, since these metrics tend to fluctuate uniquely to each test, confidence intervals of similar fashion are produced. While this data is important to see and analyze, it also can detract from message the data is trying to convey. As such, the confidence intervals are plotted both in raw true form and using the *lowess()* function in R [33]. This function uses Locally Weighted Polynomial Regression, which

creates a weight at each set of y points corresponding to a single x [34]. This allows for the linear effect that in scatterplot form is not easily seen to be revealed when the smooth line is created.

To provide examples of its effects Figure 23 shows the scatterplot of the 95% confidence interval for the controller attack. The intervals become seemingly intermixed, and it can be difficult to extrapolate the meaning of the data.

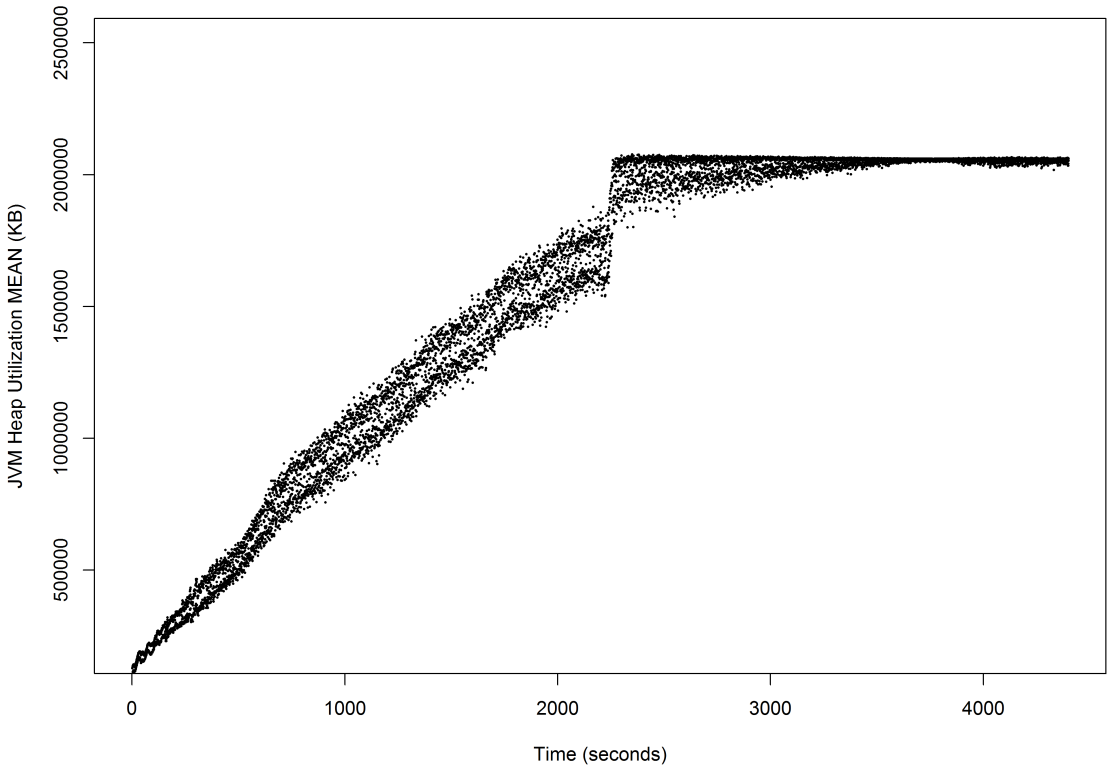

Figure 23. Example Scatterplot of 95% Confidence Interval

By changing the plot from a scatter plot to lines shown in Figure 24, the fluctuations in the intervals mentioned earlier appear. While this form is now easier to depict which data is the high interval and which data is the low interval, it leaves very little room for the mean data to be plotted. Additionally, if the intervals exhibit

the vertical fluctuations as they do, then so does the mean data from which the intervals are derived. This means once the mean data is plotted, it becomes difficult to distinguish and the graph loses the message it is trying to convey.

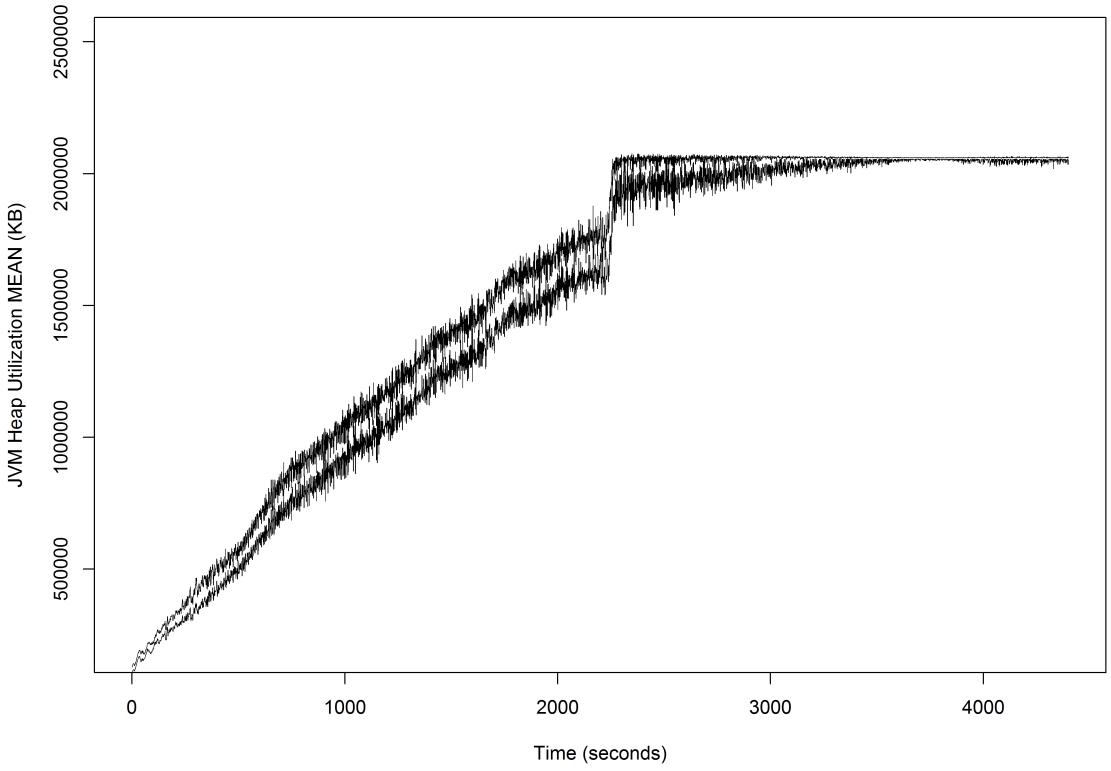

Figure 24. Example Lines of 95% Confidence Interval (No lowess() Function)

Figure 25 shows the same confidence intervals plotted with the *lowess()* function, where $f = 0.01$. The f value gives the proportion of points in the plot which influence the smoothness at each value. In other words, the larger the f value, the smoother the line and the less influence the true data has on the plot. The smaller the f value, the more influence the data has, and the closer the smoothed line is to the true data. All plots with confidence intervals going forward will have the *lowess()* function applied to them unless explicitly stated.

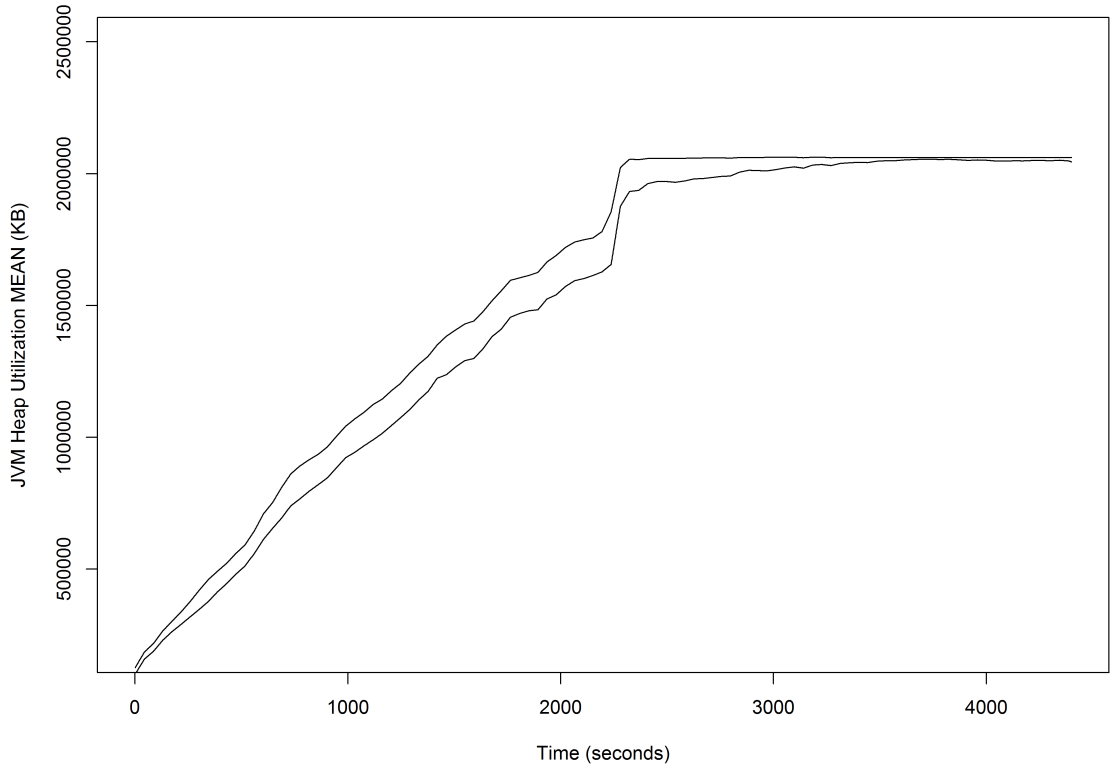

Figure 25. Example Lines of 95% Confidence Interval (lowess() Function Applied f = 0.01)

3.9 Methodology Summary

This chapter describes the methodology used to run both the switch attack experiment and the controller memory attack experiment. The experiments use different workload quantities but identical programs are executed. The switch attack experiment tests the repeatability of unforeseen hardware vulnerabilities in the Pica8 P-3290 SDN switch and sheds light on a DoS vulnerability based on how the Pica8 switch handles full flow table. The controller memory attack experiment tests to see if injecting a large amount of MSSs into the network if it is possible to cause the JVM Heap to artificially inflate and throw the *OutOfMemoryError* making the controller

program crash. Both experiments run the same set of scripts for longer or shorter periods based on the number of MSSs generated. Lastly, the evaluation techniques of how the data is analyzed was discussed. By using R to analyze and plot the data, this research is able to conclusively interpret the effect that the attack has on each experiment.

IV. Results and Analysis

4.1 Summary of Results

The experiment conducted to repeatedly trigger the discovered vulnerability in the Pica8 P-3290 SDN switch concluded with a 100% success rate and with a range of 1,366-1,457 MSSs required before the vulnerability is triggered. The controller memory attack experiment concluded with 0% of the trials producing the desired *OutOfMemoryError* despite all tests reaching the maximum JVM heap capacity. However, intermittent DoS is achieved via the Garbage Collector. Unexpectedly, the Garbage Collector completely stops performing collections in the YG and begins to exclusively perform Full Garbage Collections. Furthermore, the collection time for each Full Garbage Collection grows from taking thousandths of a second to over one second. This growth in execution time means the attack is causing the Garbage Collector to pause execution of the controller for a second at a time. The mean total time the Garbage Collector spent running across all 30 trials is 2,053.987 seconds, meaning the controller is, for all intents and purposes, offline for a mean time of 2,053.987 seconds (34.205 minutes).

4.2 Experiment 1: Pica8 Switch Attack

Figure 26 presents the number of MSSs connected to the controller when the vulnerability is triggered and are represented by the black circles on the graph. The number of MSSs needed range from 1367-1458 and with a mean of 1,412.9, represented by the horizontal blue line. These values were recorded by the author by closely monitoring the Floodlight web UI discussed in Section 3.8.7. Once the values were recorded, the actual value recorded is decreased by 1 in order to account for the Pica8 switch being connected to the controller. Only 10 trials were conducted due to the

mean data being consistently inside the 95% confidence interval.

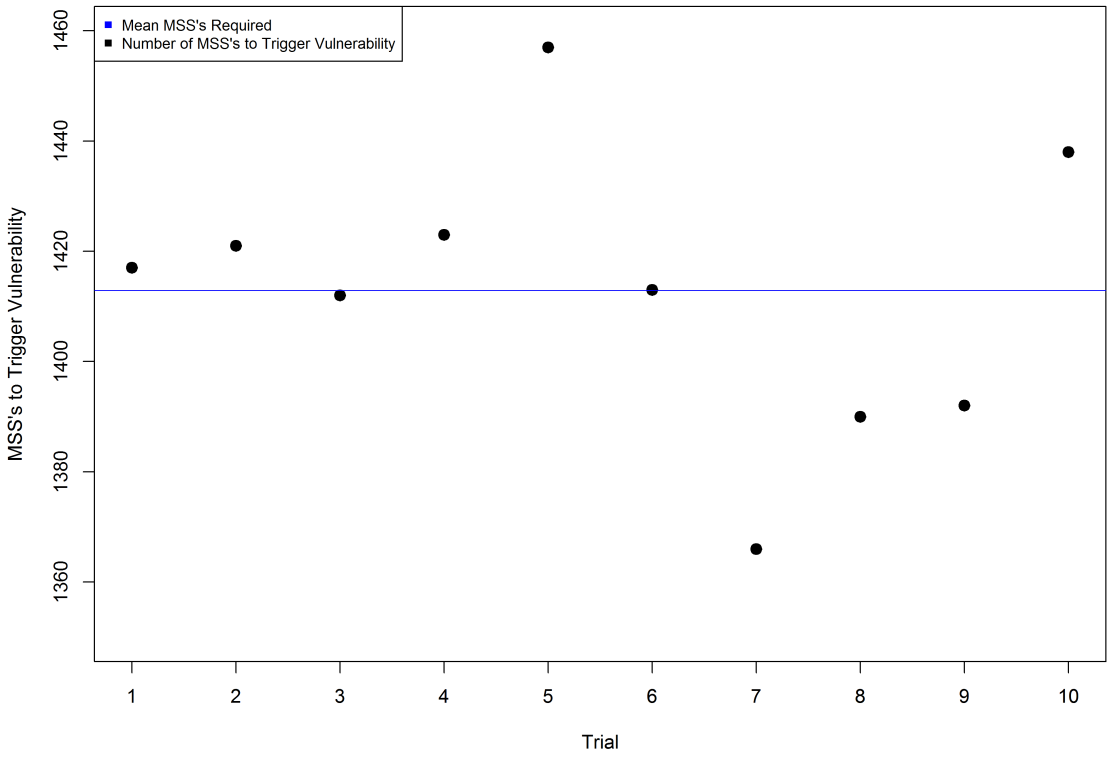

Figure 26. Experiment 1: MSSs to Trigger Vulnerability per Trial

4.2.1 Verifying Vulnerability.

Upon discovery of the vulnerability, many variables were eliminated in order to determine the cause of the vulnerability. The two variables of interest are whether or not the DoS is triggered due to connection-based or flow-based activities. In other words, is the DoS trigger due to simply too much traffic or is it because the flow tables are filled and the Pica8 does not handle this properly.

The results are conclusively that it is flow based, and the Pica8 does not follow ONF switch specifications on how to handle itself when its flow tables are full, resulting in a vulnerability that triggers a DoS. In order to verify the DoS trigger is not

connection based, static flows were pushed via Floodlight's Northbound API using the command seen in Figure 27. This command is entered on the controller's host command line terminal, where *switch_MAC* and *controller_ip* are replaced with their respective values. The command puts a permanent flow on the Pica8 switch forwarding all traffic coming in on physical port 1 to physical port 2. A second identical command was also sent with the ports reversed. This command inserts 2 static flows on the Pica8 switch, allowing for all traffic coming from the compromised host to be forwarded to the controller, and all return traffic to return back to the compromised host, resulting in no new flow entries being created during the attack.

```
curl -X POST -d '{"switch": "switch_MAC", "name":"flow-mod-1",
"priority":"1", "in_port":"1","active":"true", "actions":"output=2"}'
http://controller_ip:8080/wm/staticflowpusher/json
```

Figure 27. Static Flow Command used to Place Flows on Pica8 Switch

The attack is then executed with the same parameters seen in Table 1 in Section 3.5. Instead of a DoS being trigged and *TCP Zero-Window* packets being transmitted, all 2,000 MSSs were able to be spawned and successfully connected to the controller. The 2,000 MSSs, creates far more network traffic than what 1,367-1,458 MSSs can create ruling out the possibility that it is connection based. Lastly, the Floodlight controller configuration files were poured over rigorously to see if there are settings capable of triggering such an event, and no such settings were found, thus ruling out the controller as the cause.

In order to confirm a DoS is achieved when the vulnerability trigger point is reached and the number of connected MSSs begin to decrease, pings are sent from the compromised host to the controller, and from the controller to the compromised host, via their respective command line terminals. All pings from both sides returned with 100% packet loss with the error of *Destination Host Unreachable* despite having

created a multitude of connections with each other already. This packet loss is due to the *TCP Zero-Window* packets, being sent out by the Pica8, preventing the controller from creating the flows needed for the pings to work properly. This prevention of flow creation means, if this attack were to have taken place in an enterprise environment, new flows would not be able to be created. If new flows are not able to be created, then the network is no longer able to provide service, causing the network to fail the availability factor from the security principles discussed in Section 2.1.

The last verification step is to verify the switch does not handle its flow tables being full properly. According to the switch specifications [5] [31], the switch is supposed to send an error message to the controller with error type *OFPET_FLOW_MOD_FAILED* and with a code of *OFPFMFC_ALL_TABLES_FULL*. Inspecting the Wireshark captures, the point at which the *TCP Zero-Window* packets begin transmitting is first found and then the packets leading up to this point are searched for the error message. The error message is never found across all 10 trials confirming the switch is not following the ONF switch standards.

4.2.2 Vulnerability Trigger.

Figure 28 shows the Wireshark capture once the vulnerability is triggered and *TCP Zero-Window* packets begin transmitting. This snippet is from the trial 5 Wireshark capture and shows the switch (10.231.0.2) sending the controller (10.231.0.200) *TCP Zero-Window* packets resulting in a DoS. When this occurs, the number of connected switches shown in the Floodlight web UI (Section 3.6.7) decrease in a rapid and arbitrary fashion. This processes is how the number of MSSs required to trigger the DoS is recorded. For example, in trail 6, when the number shown by the web UI was 1,414, after this value is reached, there is a brief pause followed by a decrease down to 1,412, followed by another down to 1,411, 1,409, 1,404, and continues dropping

in arbitrary amounts with the number of switches being dropped growing in number (dropping by 10-15 at times) as the DoS continues. This drop in connected MSSs, is the signal the DoS is taking place and is one capable of observed purely from the web UI, thus in trial 6, the value of 1,413 (1,414 - 1) was recorded. The number of MSSs losing connection continue at this arbitrary rate and then begin to slow down and level off at around 900-1,000 connected MSSs.

No.	Time	Source	Destination	Protocol	Length	Info
380815	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	182	[TCP ZeroWindow] Type: OFPT_PACKET_IN
380816	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	182	[TCP ZeroWindow] Type: OFPT_PACKET_IN
380817	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	182	[TCP ZeroWindow] Type: OFPT_PACKET_IN
380818	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	182	[TCP ZeroWindow] Type: OFPT_PACKET_IN
380819	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	174	[TCP ZeroWindow] Type: OFPT_PACKET_IN
380820	1688.19…	10.231.0.2	10.231.0.200	OpenFlow	182	[TCP ZeroWindow] Type: OFPT_PACKET_IN

Figure 28. Experiment 1: TCP Zero Window Packet Transmission from Switch

When the DoS is triggered, the Pica8 switch still maintains many of its flows, yet, MSS connections drop considerably over a period of hundreds of seconds. inspecting the Floodlight Controller Logs for each trial, there is a clear section in the logs where switches are being listed as disconnected; however, no further explanations are provided for the dropped switches. This behavior can been seen in Figure 29, which is a snippet taken from one of the log files from the trials. The red box highlights what the controller records in its logs when a switch is disconnected, showing that the controller does not provide any indication as to reasoning behind the disconnects. There are a few reasons as to why the controller is dropping roughly 1/3 of the MSSs and are discussed in the following subsections.

```
[[e8:d9:3d:cf:65:4f:7d:3d(0x0) from 10.231.1.201:42860]] Disconnected connection
[[f7:6b:db:7c:49:33:a4:67(0x0) from 10.231.1.201:33454]] Disconnected connection
 [[c7:68:06:89:c2:50:07:05(0x0) from 10.231.1.201:34838]] Disconnected connection
 [[ce:ea:1a:08:e8:9c:10:7f(0x0) from 10.231.1.201:34564]] Disconnected connection
```

Figure 29. Experiment 1: Controller Log File During DoS

4.2.2.1 Handshake Complete with Auxiliary Messages in Transit.

As described in Section 2.3.5, once the controller/switch handshake is complete, the switch (MSS) is now a registered switch with the controller. However, more OFP messages after the handshake completion are sent. These messages are both requests from the controller and the respective replies from the switch intended to provide deeper details into the switch's capability, flow tables, directions on how to handle packet fragmentation, etc. When these requests are sent, if the controller does not receive the proper reply message from the switch or receives nothing at all, it begins sending *OFPT_ECHO_REQUEST* messages to the switch with the expectation of an *OFTP_ECHO_REPLY*. This exchange of echo request/reply is used by the controller in order to verify the connection is still valid and working properly. If five request/reply messages are successfully sent/received and the original auxiliary message request does not receive its proper reply by the fifth request/reply message, the controller deems the connection to be dead and drops the switch. When the DoS is occurring and *TCP Zero-Window* packets being sent, neither the switch nor controller are able to send the messages needed to complete the exchange of auxiliary messages and thus switches (MSSs) result in being dropped. Since these auxiliary messages are unable to be sent, their corresponding request/reply message, which will be sent due to the axillary message reply not being received, are also unable to be sent.

4.2.2.2 Echo Request not Returned.

As mentioned, *OFPT_ECHO_REQUEST* messages are sent from the controller to the switch with the expectation of an *OFPT_ECHO_REPLY* being sent back, and are transmitted in order to verify the connection. These messages are not just sent when auxiliary messages are not properly exchanged, they are also sent when there

have not been any OFP messages exchanged by the controller/switch. The ONF specifications do not state an explicit amount of time before this exchange begins; however, it was observed during testing to be immediately after the last query message was sent by the Floodlight controller. The controller constantly sends echo requests and expects echo replies until a OFP message is exchanged between controller and the respective switch. The controller does this exchange with every switch (MSS) connected to it. This process means, if 1,300 MSSs are connected, the controller is attempting to exchange echo request/reply messages with all of them. It is likely when the vulnerability is triggered, despite already existing flows being on the switch, the echo replies are getting dropped by the Pica8 switch. If these replies are never received by the controller, the connection is considered dead and the connection is dropped.

It is also important to note that the attack tool (MSS) does not care if it receives these echo requests from the controller and will continue to maintain a connection to the controller despite not receiving the echo request messages. Therefore, despite the controller being unable to send the echo request messages due to the *TCP Zero-Window* packets being sent, on the MSS side of the connection, the tool does not initiate a disconnect and continues to maintain the connection until the controller initiates the disconnect. Moreover, the MSSs also do not send echo replies to the controller since no corresponding echo request has been received.

4.2.2.3 Connection Timeout.

The OFP is an application layer protocol that rides atop TCP. Therefore, if a reply to an echo request, or any other auxiliary messages made it to the controller or switch before the DoS is triggered, at the minimum the sender is expecting a TCP *ACK* in response, at some point. If these *ACKs* are unable to make it back, the

connection will eventually timeout. This timeout could occur on the compromised host side or the controller side.

4.2.2.4 Pica8 Switch Removing Flows.

It is also possible the Pica8 switch may be slowly removing flows in order to free up space within its flow tables. This flow removal is a possible explanation, but cannot be confirmed since knowledge of exactly how the Pica8 is designed and programmed is not public knowledge. However, if the switch already does not follow the standards set forth by the ONF, it is possible the switch operates outside the bounds of the standard and deletes flows when its tables are full in order to create room.

4.2.3 Controller View.

Figure 30 shows the mean JVM Heap Utilization for all 10 trials, and Figure 31 shows the same data with the smoothing function applied. The top blue line and the bottom blue line are the high and low 95% confidence intervals respectively, the black line is the mean heap utilization, and the red line is the best fit line of the mean data. The 95% confidence intervals are plotted with f-value of 0.03, and the best fit line is plotted with an f-value of 0.2. A low f value was selected for the confidence interval in order to provide an adequate amount of smoothing while not injecting bias into the data. On the other hand, for the best fit line, a smoother line is required and thus an f-value of 0.2 is used. The mean data plot is left untouched and is in its original state in both Figure 30 and 31.

As mentioned in Section 4.2.2, the rate of MSSs that disconnect begins to level off between 900-1000 connected switches. The effects of this leveling off can be seen in Figure 31 towards the right end of graph. The confidence intervals, mean, and best fit line, stop growing linearly and begin to slope downward at approximately 1,700

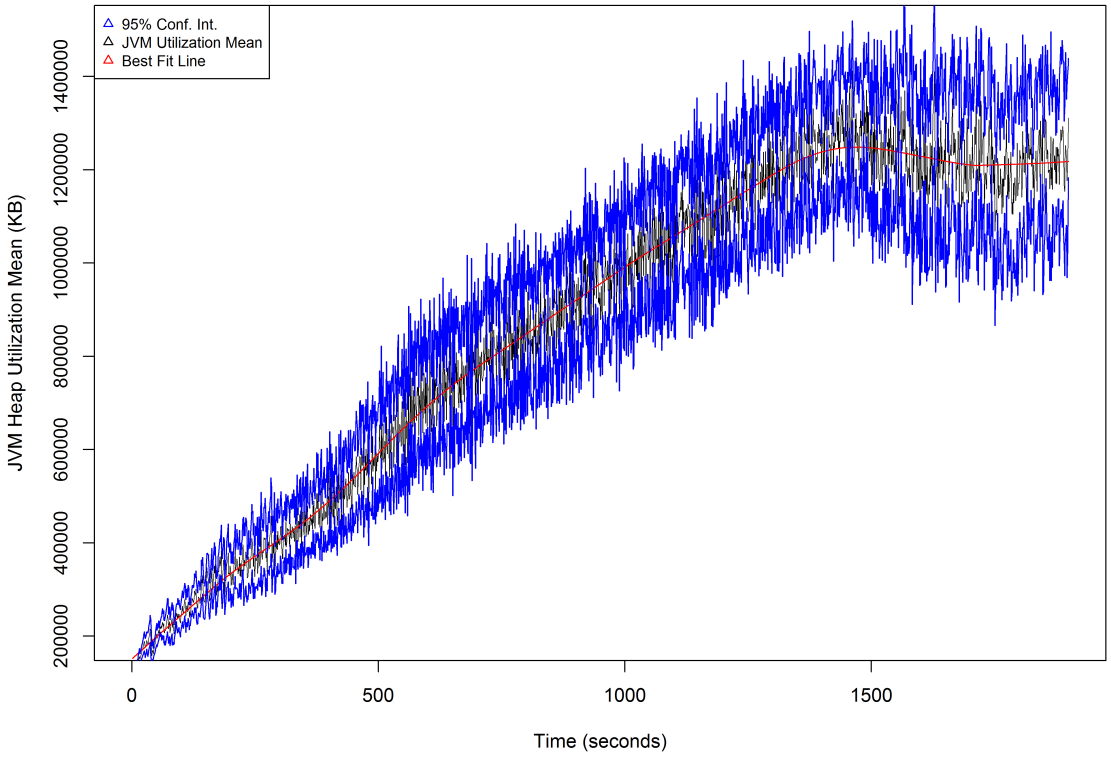

Figure 30. Experiment 1: Switch Attack Mean JVM Heap Utilization No Smoother

seconds the confidence intervals, mean, and best fit line, begin to turn upwards again as service begins to return to the network, resulting in MSSs being able to reconnect to the controller. Supplemental graphs for Experiment 1 can be found in Appendix B.

4.3 Experiment 2: Controller Memory Attack

The controller memory attack concluded with 0% of the 30 trials producing the desired Java *OutOfMemoryError*, and thus, the attack did not cause the controller to crash. However, intermittent DoS is achieved by causing the Garbage Collector to flip from mainly performing YG Garbage Collections to exclusively performing

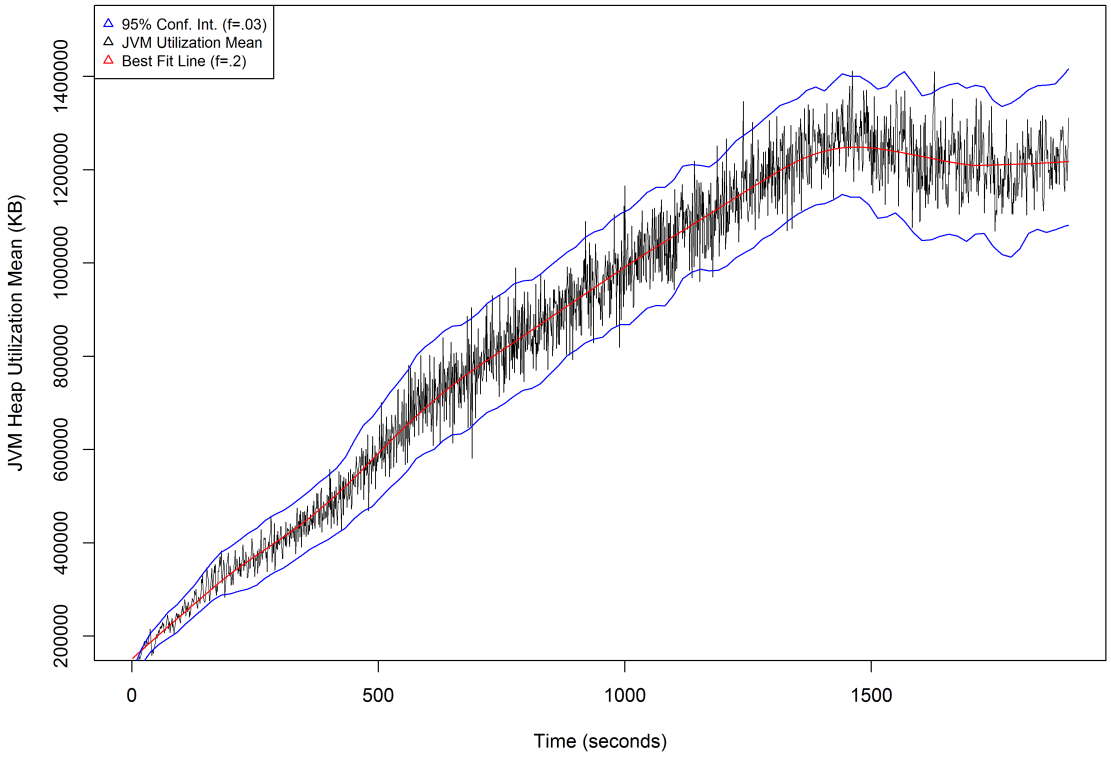

Figure 31. Experiment 1: Switch Attack Mean JVM Heap Utilization Smoother Applied

Full Garbage Collections. Additionally, due to the raw amount of memory these Full Garbage Collections have to parse through, Full Garbage Collection time is increased from the normal run time of thousandths of a second to over one second. The max value the mean capacity reaches is 2,081,250.8 KB (2.08125 GB). This value is 99.64% of the maximum JVM Heap Capacity (2,088,763.39 KB/2.0887634 GB) allowed for a host configured with 8 GB of memory. Figure 32 displays the command used on the controller's host to obtain important JVM Heap-space default values, which are returned in bytes. Table 2 shows the returned settings that are pertinent to this research. A screen shot of the terminal command and the full response can be found in Appendix A while supplemental graphs for Experiment 2 can be found in Appendix

B. The following sections discuss these results in further detail.

```
java -XX:+PrintFlagsFinal -version |grep -iE 'HeapSize|PermSize|ThreadStackSize'
```

Figure 32. Command to Access JVM Default Parameters

Table 2. Important JVM Heap-space Defaults

JVM Setting	Setting Value (bytes)	Setting Value (gigabytes)
InitialHeapSize	132120576	0.132120576
MaxHeapSize	2088763392	2.088763392

4.3.1 JVM Heap Capacity Results.

Figure 33 shows the mean JVM Heap Capacity across all 30 trials with no smoothing function applied. Figure 34 shows the same data with the smoother function applied to the 95% confidence interval where the f-value is 0.01. A very small f-value is chosen in order to prevent bias being injected into the graphs, yet the value provides enough smoothing to allow for the meaning of the data to still be extrapolated. The plotted data shows the mean stays tightly within the confidence intervals until a point in which it is clear some data deviates outside of these intervals due to the confidence intervals converging.

The points that deviate outside of the intervals after 2,320 seconds do so simply because during this time the trials are bumping into the maximum allowed capacity. Since the JVM can no longer dynamically grow its heap, it is forced to rely on Garbage Collections in order to maintain its size. If the Garbage Collector happens to clear enough memory, the JVM may shrink a specific generation's capacity in an effort to back away from the memory ceiling it has reached. This is why points fall outside the

confidence intervals the way they do; however, it is also clear that fewer and fewer points fall outside the confidence intervals as time goes on..

Despite not throwing the *OutOfMemoryError*, this data is significant because it shows the attack is successful in causing the JVM Heap to grow to its maximum bounds. Reasons why this growth did not continue are discussed in later sections.

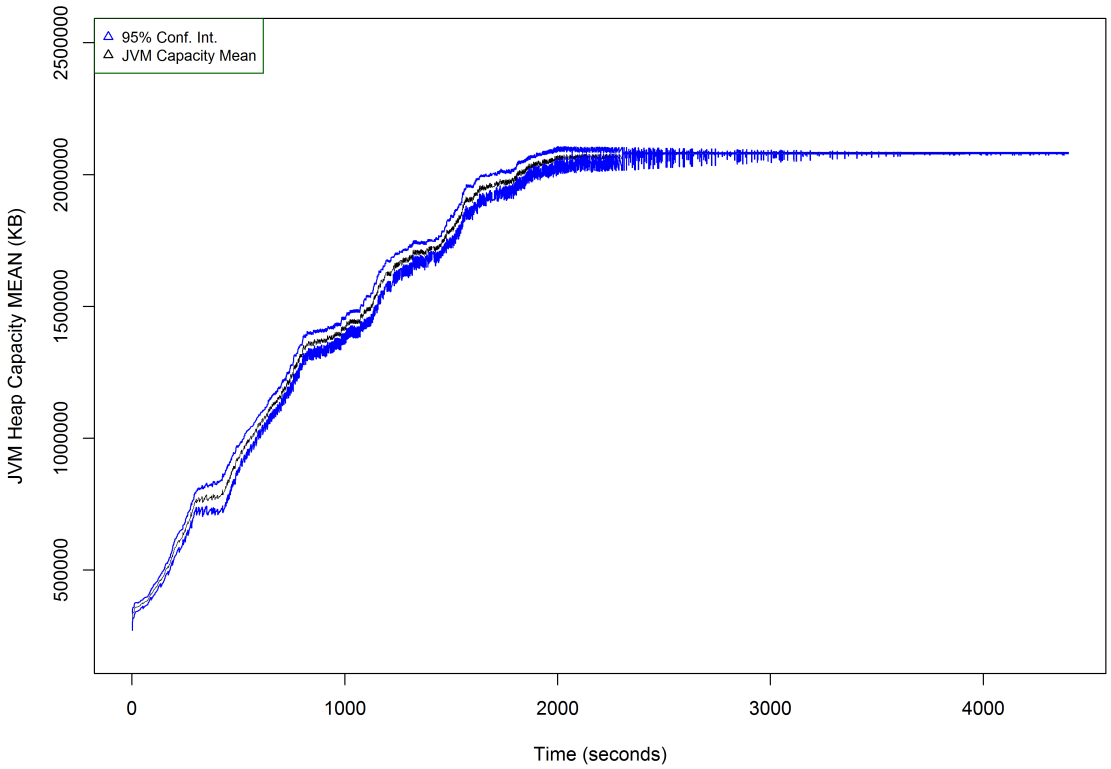

Figure 33. Experiment 2: Controller Memory Attack Mean JVM Heap Capacity No Smoother

4.3.2 JVM Heap Utilization Results.

Figure 35 shows the mean JVM Heap Utilization across all 30 trials with no smoothing function applied. Figure 36 shows the same data with the smoother function applied to the 95% confidence interval where the f-value is 0.01. A very small

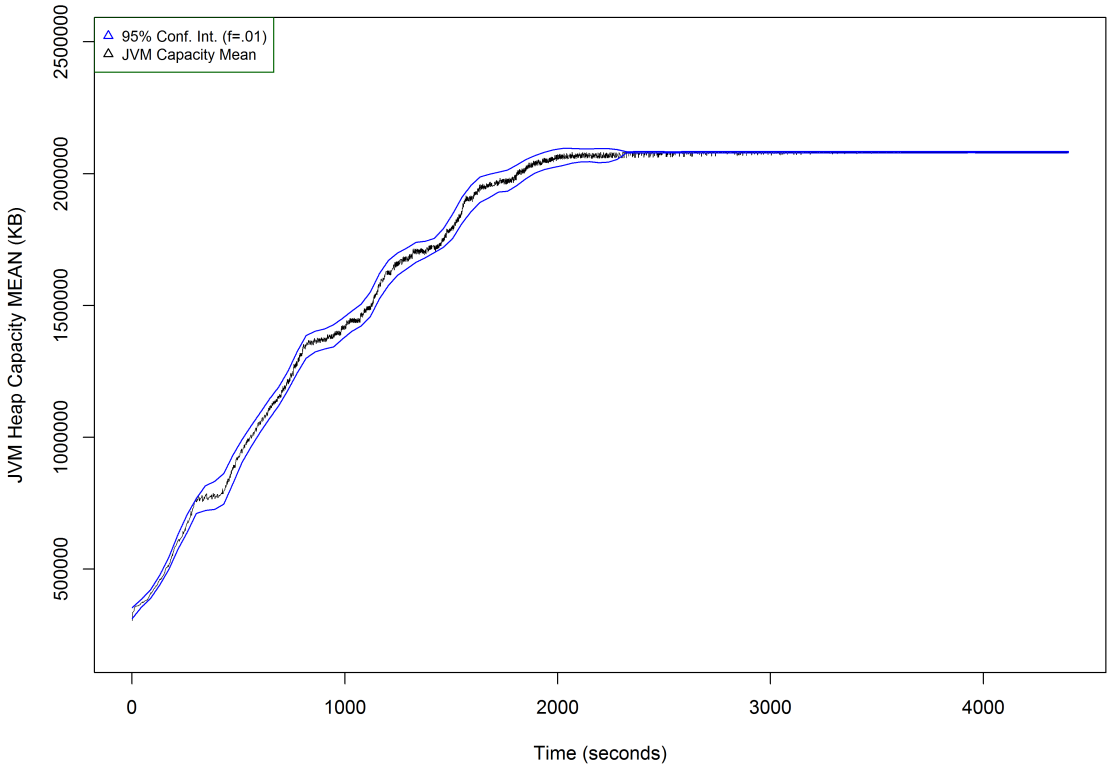

Figure 34. Experiment 2: Controller Memory Attack Mean JVM Heap Capacity Smoother Applied

f-value is chosen in order to prevent bias being injected into the graphs, yet the value provides enough smoothing to allow for the meaning of the data to still be extrapolated. The mean data plotted stays well within the bound of the confidence intervals with very few points deviating outside the bounds. This shows the attack can consistently consume large amounts of the JVM's available heap space, with the maximum of the mean utilization being 2,058,195.0 KB (2.08125 GB). This maximum is 98.53% of the maximum allowed size the JVM can grow its heap to which is just below the heap ceiling discussed in Section 4.3.1.

Inspecting Figure 36, there is a notable "jump" in the mean utilization. At first this jump may seem arbitrary; however, there is a very important event occurring at

this point within the JVM. It is at this point that the JVM switches from performing mostly YG Garbage collections to exclusively Full Garbage Collections. More details on this particular event and its importance are discussed in the next section.

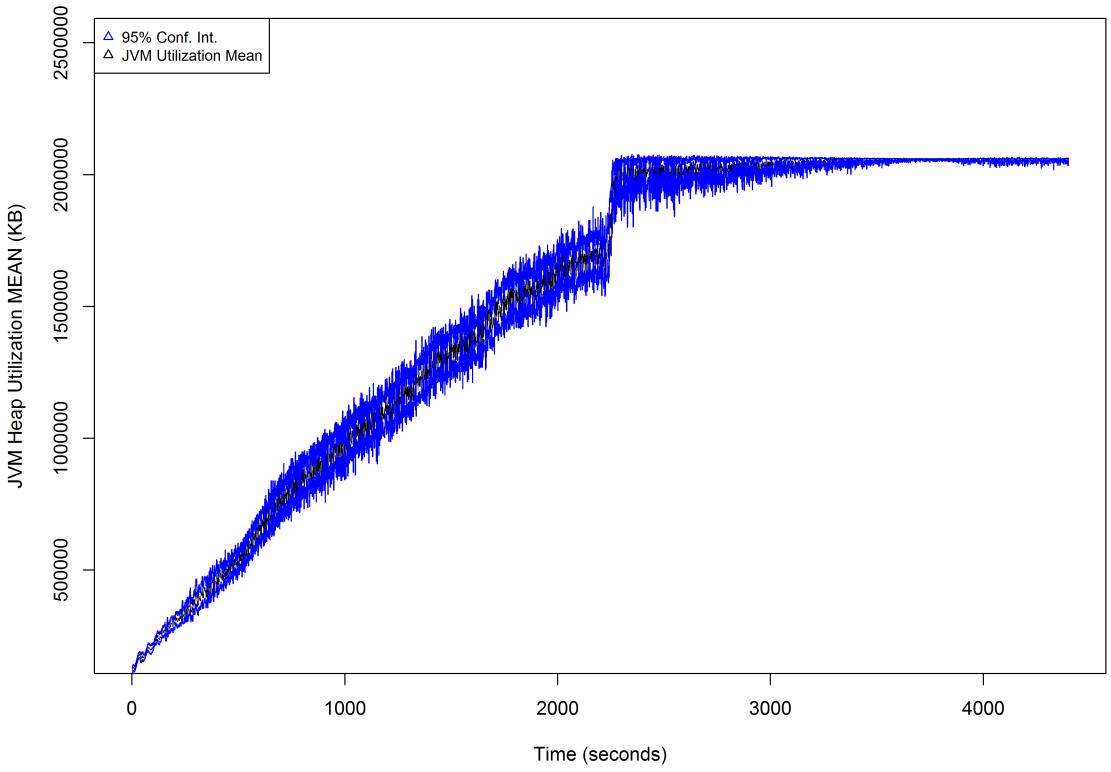

Figure 35. Experiment 2: Controller Memory Attack Mean JVM Heap Utilization No Smoother

4.3.3 JVM Garbage Collection Results.

4.3.3.1 Young Generation Garbage Collection.

Figure 37 shows the mean number of YG Garbage Collections performed across all 30 trials with a mean of 1,459 YG Garbage Collections performed. The 95% confidence interval is drawn with no smoothing function applied due to the fact that when plotted the confidence interval is already smooth and easily readable. The

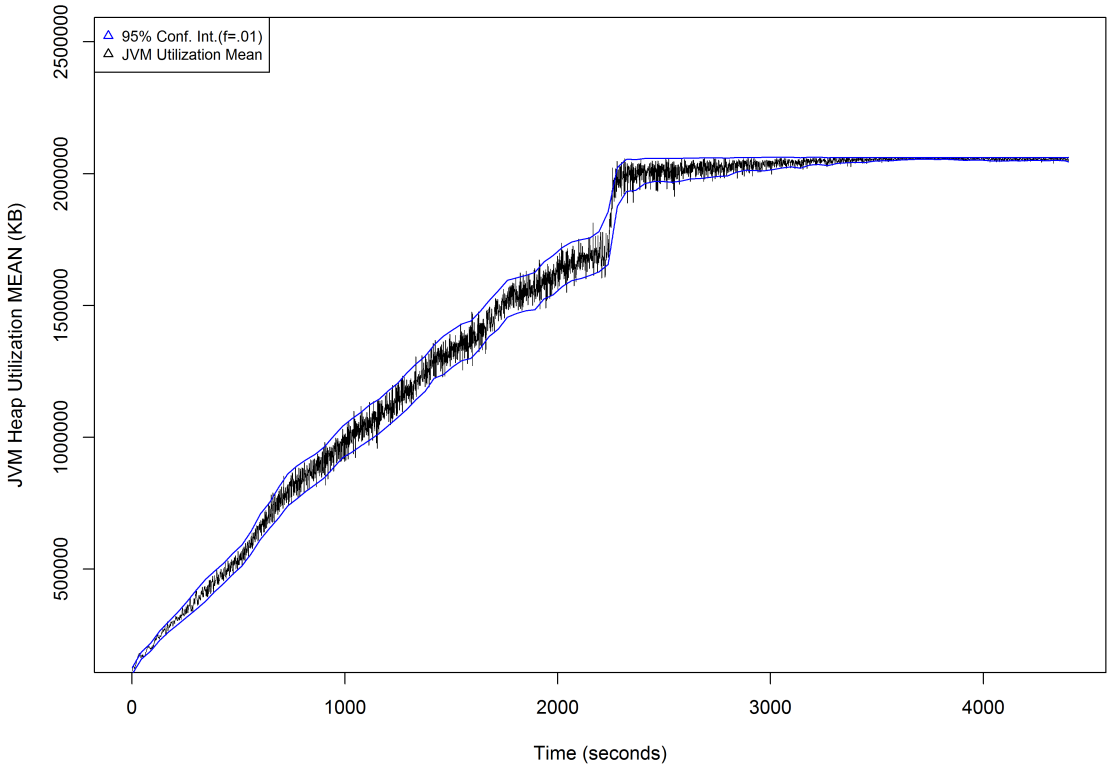

Figure 36. Experiment 2: Controller Memory Attack Mean JVM Heap Utilization Smoother Applied

mean data stays within the confidence interval bounds 100% of the time, showing the behavior of the YG Garbage Collector can be predicted with a high degree of confidence. There is a point in which the mean data and the confidence intervals hit a max point and then level off without further increases for the duration of the graph. This is the point where the Garbage Collector is switching to performing Full Garbage Collections exclusively.

4.3.3.2 Full Garbage Collection.

Figure 38 shows the mean number of Full Garbage Collections performed across all 30 trials with a mean of 2,156 Full Garbage Collections performed. The 95%

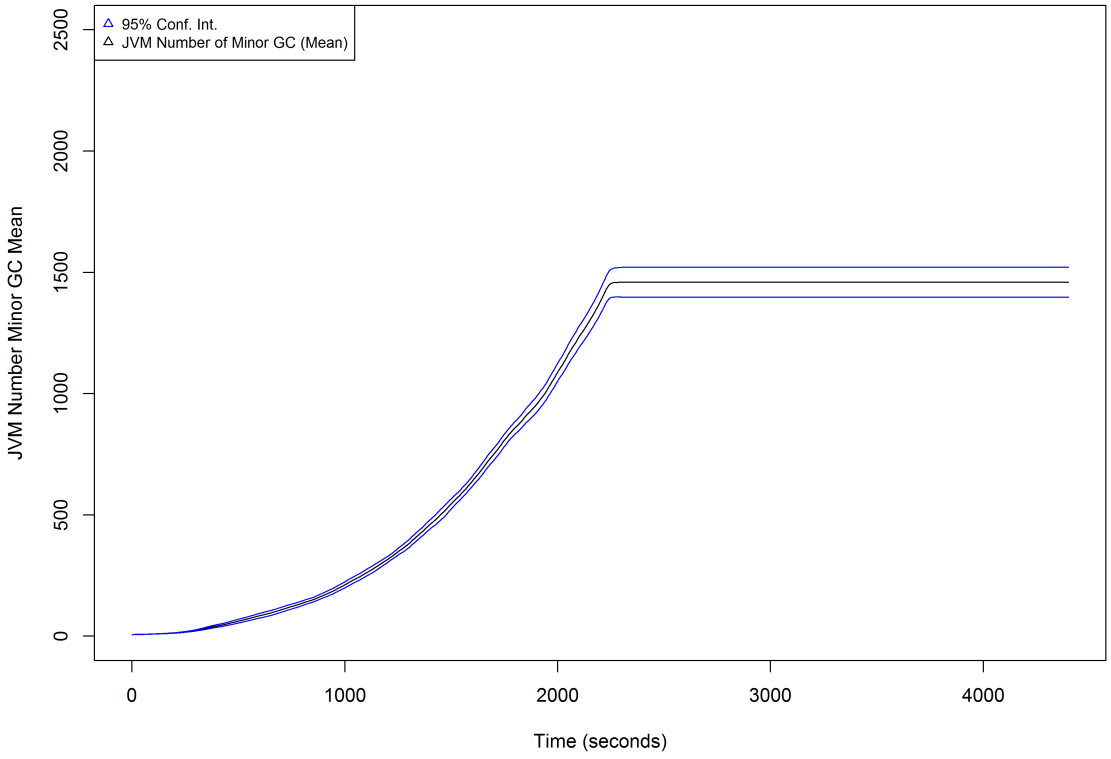

Figure 37. Experiment 2: Controller Memory Attack Mean Number of JVM YG Garbage Collections No Smoother

confidence interval is drawn with no smoothing function applied due to the fact that when plotted the confidence interval is already smooth. The mean data stays within the confidence interval bounds 100% of the time with the confidence intervals being so tight the mean data and intervals become a singular line for much of the graph. Even more so than the YG Garbage Collection results, the Full Garbage Collection behavior is extremely predictable. There are two areas of focus for this data. The first and most obvious is the point at roughly $x = 2{,}200$, where the Full Garbage Collections begin to run exclusively, thus the number of Full Collections begin to increase rapidly.

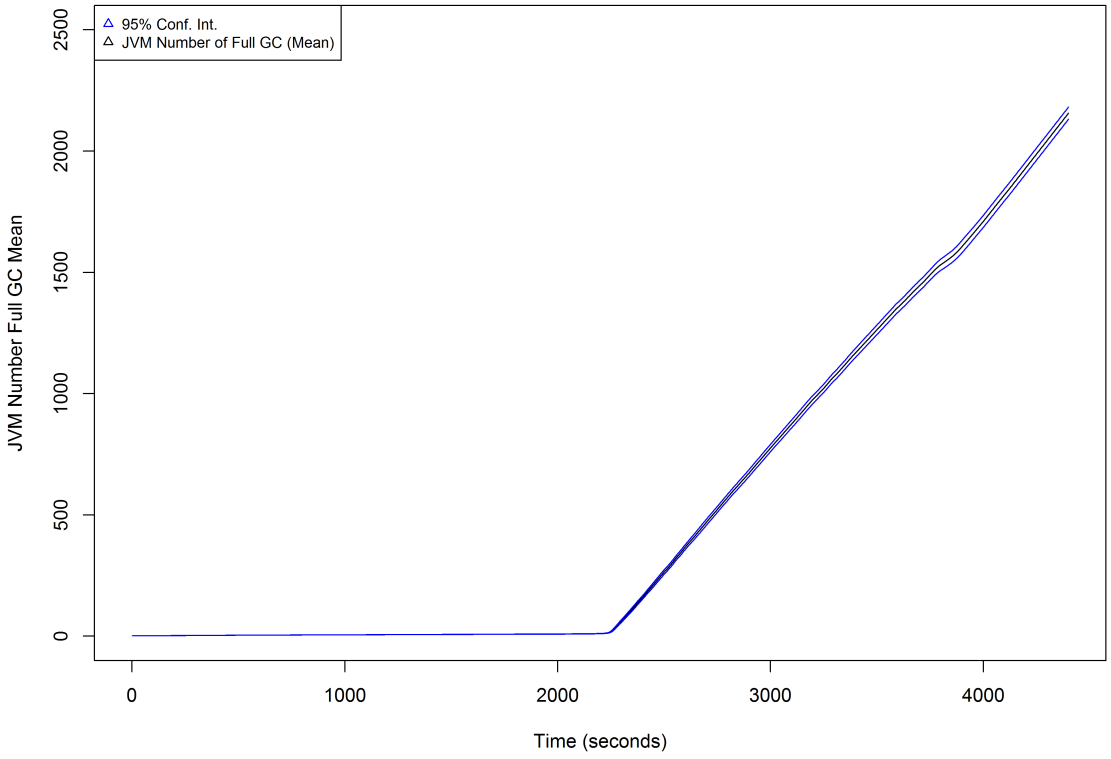

Figure 38. Experiment 2: Controller Memory Attack Mean Number of JVM Full Garbage Collections No Smoother

4.3.3.3 Root of Attack Failure.

The second point of interest in Figure 38, occurs at around $x = 3{,}800$, where the linear behavior of the line flattens out slightly only to begin increasing again. Since the Full Garbage Collections begin to take full seconds to complete and Garbage Collection events are STW events, this means the controller is not running for full seconds at a time. Due to the prolonged Garbage Collection times and sheer quantity of collection events, the attack was unable to proceed past approximately 3,250 MSSs. This slight dip is a point in which the controller begins to lose connections to many of the MSSs (100-200). The Garbage Collector then clears the old unreferenced objects that are no longer in use and then slightly backs off in its frequency of execution since

memory has been freed. This effect can be observed in Figure 35, there the confidence intervals begin to expand slightly at around the $x = 3{,}800$ mark.

4.3.3.4 Total Garbage Collection Time.

As mentioned in Section 4.1, intermittent DoS is achieved by unforeseen Garbage Collection performance. Due to the attack causing the Garbage Collector to spend large quantities of time cleaning the heap, this causes the controller to be halted in its execution for the same amount of time. For example, if the Garbage Collector spent 2,000 seconds in total, then the controller spent 2,000 seconds in total not executing. If the controller is not executing then, for all intents and purposes, the network is not able to provide service. New flows are not be able to be created until the controller resumes execution, and when it does resume execution, there is no guarantee how long it will last before another Full Garbage Collection is triggered.

Figure 39 shows the mean total time JVM spent Garbage Collection throughout all 30 trials. The 95% confidence interval is drawn with no smoothing function applied due to the fact that when plotted the confidence interval is already smooth. The mean data stays within the confidence interval bounds 100% of the time with the confidence intervals being so tight the mean data and intervals become a singular line for nearly the entire graph. Again, this small standard error signifies the total amount of time spent garbage collecting is nearly the same across the trials and thus can confidently be predicted if further trials are performed. The maximum of the mean is 2052.287 seconds, which as mentioned previously, means the controller is offline for a mean time of 2052.287 seconds (34.205 minutes). The amount of time the controller spent in a suspended state was not consecutive and was spaced out with the majority of the time occurring approximately after $x = 2200$.

Interestingly enough, the point at which the time begins to move aggressively

upward is also the same point in which the number YG Garbage Collections flatten out (Figure 37) and the same point in which the number Full Garbage collections begin to grow. The next section performs a comparison of these three data sets and peers into how they all relate.

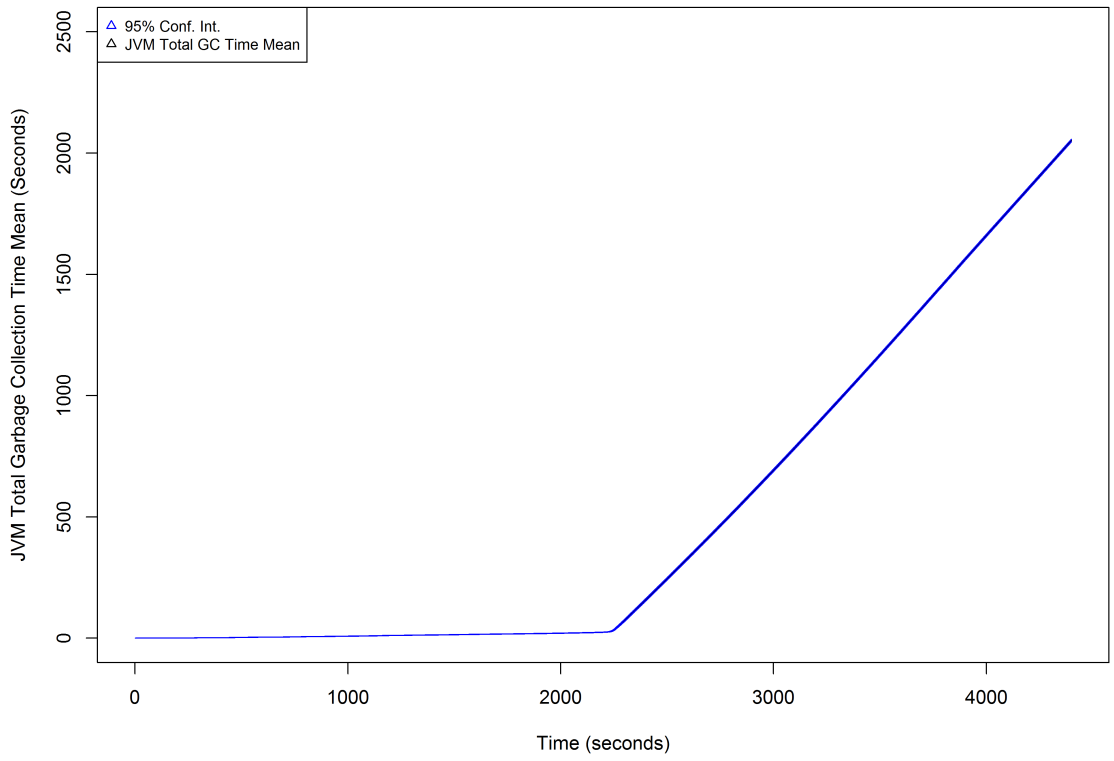

Figure 39. Experiment 2: Controller Memory Attack Mean Total Garbage Collection Run Time No Smoother

4.3.4 Confirming Garbage Collection Switch.

As discussed throughout Section 4.3, there is a point in which the JVM switches to exclusively performing Full Garbage Collections. This switch has an effect that can be see across multiple data sets and in the end results in being the root cause of failure for the attack. When the mean data from Figures 37 and 38 are plotted on the

same graph, it is possible to draw a vertical line at $x = 2{,}238$ *seconds* which signifies when the transition to Full Garbage Collections occurred. This transition can be observed in Figure 40, where the blue line is the mean YG Garbage Collections, the black line is the mean Full Garbage Collections, and the vertical red line is marking the point of the transition. This figure confirms that the transition to the exclusive performance of Full Garbage Collections is detectable through multiple data sets.

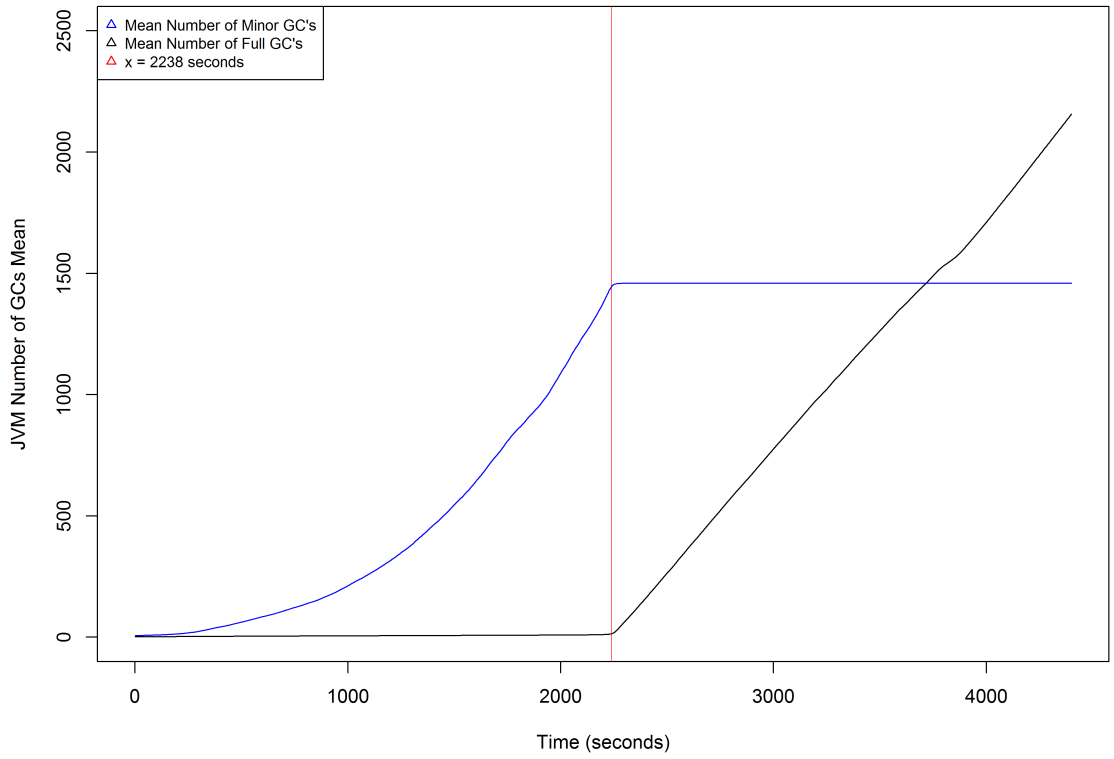

Figure 40. Experiment 2: Controller Memory Attack YG Collections versus Full Collections

Figures 41 and 42 show what happens if the same vertical line at $x = 2{,}238$ *seconds* is added to the Total Garbage Collection Run Time and the JVM Heap Utilization. A clear pattern can now be observed across a total of 4 data sets, showing truly how much of an effect exclusive Full Garbage Collections had on the 30 trials.

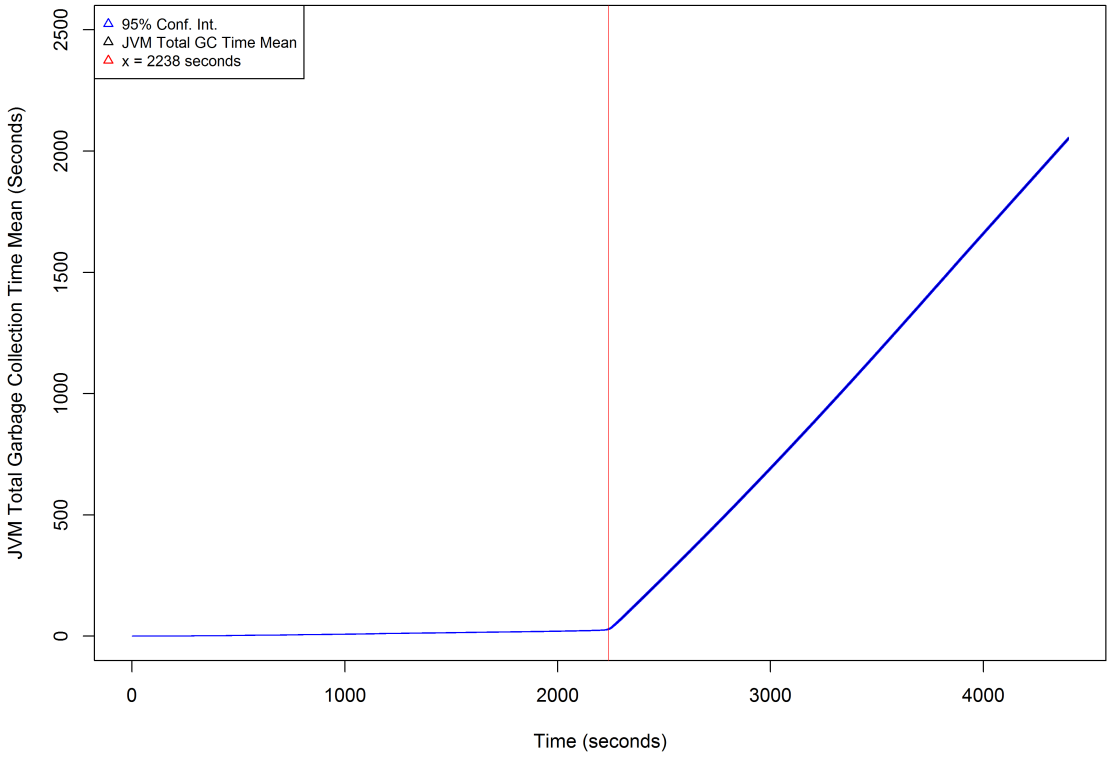

Figure 41. Experiment 2: Controller Memory Attack Mean Total Garbage Collection Run Time With Line

4.4 Results and Analysis Summary

During Experiment 1, it is seen that the attack to trigger the vulnerability on the Pica8 switch is able to be repeated with a 100% success rate, and while this particular test may not be the fastest, it is able to confirm the vulnerability as well as shed light in what really triggers the DoS.

During Experiment 2, it is seen that the number of connected MSSs cannot move pass the soft cap of 3,250, the attack is not able to push the utilization to the point required to crash the controller. If the Garbage Collector had not run rampant and begin to consume full seconds, the attack would have likely been successful. The growth trend of heap memory utilization is visibly upward reaching on average 98.53%

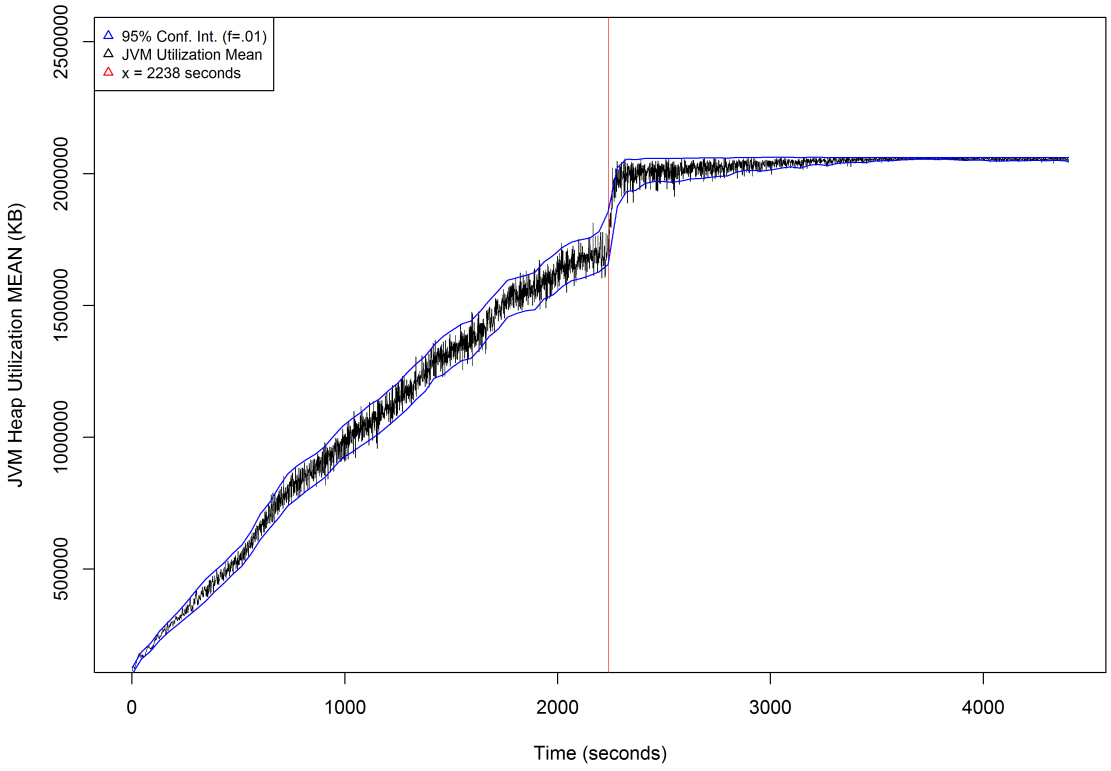

Figure 42. Experiment 2: Controller Memory Attack Mean JVM Heap Utilization With Line

of the maximum allowed JVM heap size, while the capacities on average reached 99.64% of the maximum JVM Heap Capacity. The amount of time the Garbage Collector keeps the controller offline is too long, and thus, MSSs lose connection. While the desired *OutOfMemoryError* is not achieved, the Garbage Collector does cause the controller to be offline for mean time of 2,052.287 seconds (34.205 minutes). This duration is a rather substantial amount of time, and if it had occurred in an enterprise environment would cause notable intermittent DoS.

V. Conclusions and Recommendations

This chapter summarizes the research performed and discussed throughout this thesis. Sections 5.1 and 5.2 present the research conclusions derived from the results presented in this thesis. Section 5.3 discusses the impact of this research in the area of SDN security. Lastly, Section 5.4 presents potential future work in this field.

5.1 Research Conclusions: Switch Attack

This research discovered an unknown vulnerability present in the Pica8 P-3290 SDN switch. According to the ONF switch specifications, when a switch's flow tables are full, the switch is suppose to send an error message to the controller with error type *OFPET_FLOW_MOD_FAILED* and with a code of *OFPFMFC_ALL_TABLES_FULL* [5] [31]. The Pica8 does not perform this function and instead, sends *TCP Zero-Window* packets to the controller. This vulnerability, if triggered on a enterprise network, would prevent the controller from being able to exert control over the switch as it is designed, resulting in a DoS.

5.2 Research Conclusions: Controller Memory Attack

As hypothesized, by connecting a large number of MSSs to the controller, the utilized JVM Heap memory is inflated. However, the hypothesis is incorrect about what would result from this memory inflation. Instead of the *OutOfMemoryError* being thrown and the controller crashing, the Full Garbage Collector begins running for prolonged periods of time (exceeding 1 second), as well, the Full Garbage Collector is executed at a high frequency, with a mean of 2,156 executions, resulting in intermittent DoS. This DoS occurs as a by-product of the Garbage Collections, since all Garbage Collections are STW events. Due to the Garbage Collector's increase

in execution frequency and duration of its collections, it results in a mean time of 2052.287 seconds (34.205 minutes) the controller spends in a suspended state. If the controller is halted in execution, it is unable to exert control and instruct its switches on how to forward network traffic, resulting in the aforementioned DoS. This behavior of the Garbage Collector is the root cause of the attack failing to trigger the *OutOfMemoryError*. Due to the frequency and durations of the Full Garbage Collector's executions, it causes MSSs to drop connection to the controller. This results in the utilization being unable to reach the level required to trigger the *OutOfMemoryError* and crash the controller.

5.3 Significance of Research

5.3.1 Research Contributions.

This research provides a vulnerability disclosure and insight into what a malicious attacker can do if a host were to be compromised on the network. Vulnerability discoveries in a networking paradigm as young as SDN are important in helping secure the system as a whole. Vendors will now know the results of not following the standards set forth by the Open Networking Foundation can lead to a DoS. Networking vendors do not want to be the reason for causing a DoS, since availability is one of the core security pillars they are supposed to provide. This research has also shown controllers programmed in the Java programming language need to be aware attackers may try to exploit its limited memory growth potential. As well, despite the attack not achieving the expected results, DoS is still achieved, even if it is only for seconds at a time, these periods the controller is spends in a suspended state add up, with a mean total time of 34.205 minutes. As mentioned, it is very difficult to build security into a system or protocol if the developer is not aware of what threats need to be defended. This research provides insight into an "out-of-the-box" approach to

attacking SDN, thus providing knowledge of what developers need to defend against.

5.4 Recommendations for Future Work

5.4.1 Alternative Vulnerability Triggering Methods.

Since the vulnerability trigger is caused due to the flow tables becoming full and the switch mishandling this event, it is important to note that this attack can be perpetrated by much simpler means such as a UDP flood. This test in particular was conducted due to the fact this method is how the vulnerability was discovered, and it is necessary to verify the repeatability of the method used during discovery. However this method is certainly not the only method nor is it the fastest method to trigger the vulnerability. The controller's lack of ability to handle simultaneous connections from switches is a limiting factor in the speed of which this attack can trigger the DoS. If the compromised host were to send a flood of UDP packets to randomized IPs, this attack would likely be able to be completed in a couple minutes rather than the nearly thirty minutes this test took. This style of attack is a promising area for future research.

Other areas of future research include measuring the level of network degradation that occurring when the Garbage Collector is keeping the controller offline for as long as it does. Another area of future work, is to see if there is another attack vector capable of causing the desired *OutOfMemoryError* to be thrown, while at the same time, preventing the Garbage Collector from causing the attack to fail. This attack vector would likely have to be a method not reliant on network connectivity in order to survive. This realm of research is important in helping secure the next generation of networking infrastructure, and as such needs to be continued in order to prevent the security pitfalls plaguing traditional networks for so many years.

5.5 Chapter Summary

In summary, this chapter presents the research conclusions and significance for this research. Additionally, the repercussions of the disclosed vulnerability and the results of the attack on the controller allows vendors to incorporate security into their respective products to defend against such attacks.

Appendix A. Java Virtual Machine Defaults

```
adogg@ubuntu:~$ java -XX:+PrintFlagsFinal -version | grep -iE 'HeapSize|PermSize|ThreadStackSize'
     intx CompilerThreadStackSize                  = 0                                   {pd product}
    uintx ErgoHeapSizeLimit                        = 0                                   {product}
    uintx HeapSizePerGCThread                      = 87241520                            {product}
    uintx InitialHeapSize                         := 65011712                            {product}
    uintx LargePageHeapSizeThreshold               = 134217728                           {product}
    uintx MaxHeapSize                             := 1033895936                          {product}
     intx ThreadStackSize                          = 1024                                {pd product}
     intx VMThreadStackSize                        = 1024                                {pd product}
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)
adogg@ubuntu:~$
```

Figure 43. Screenshot of JVM Defaults for both Experiments

Appendix B. Switch Attack Additional Graphs

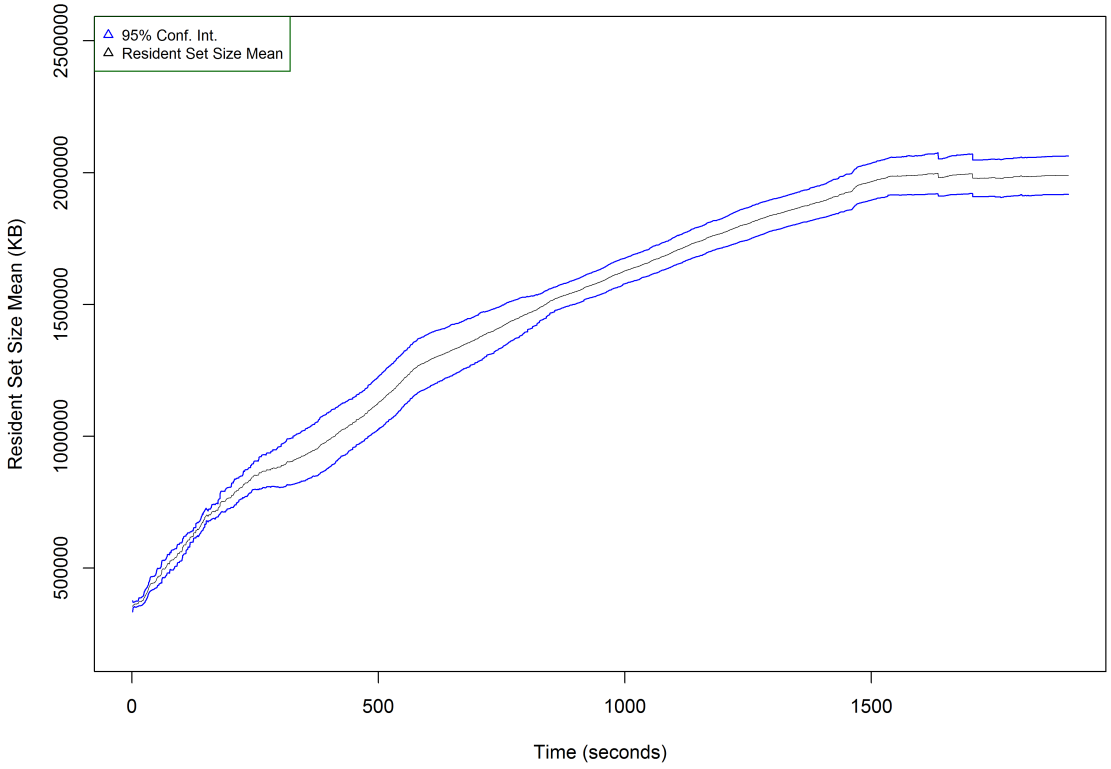

Figure 44. Experiment 1: Switch Attack Mean RSS Memory

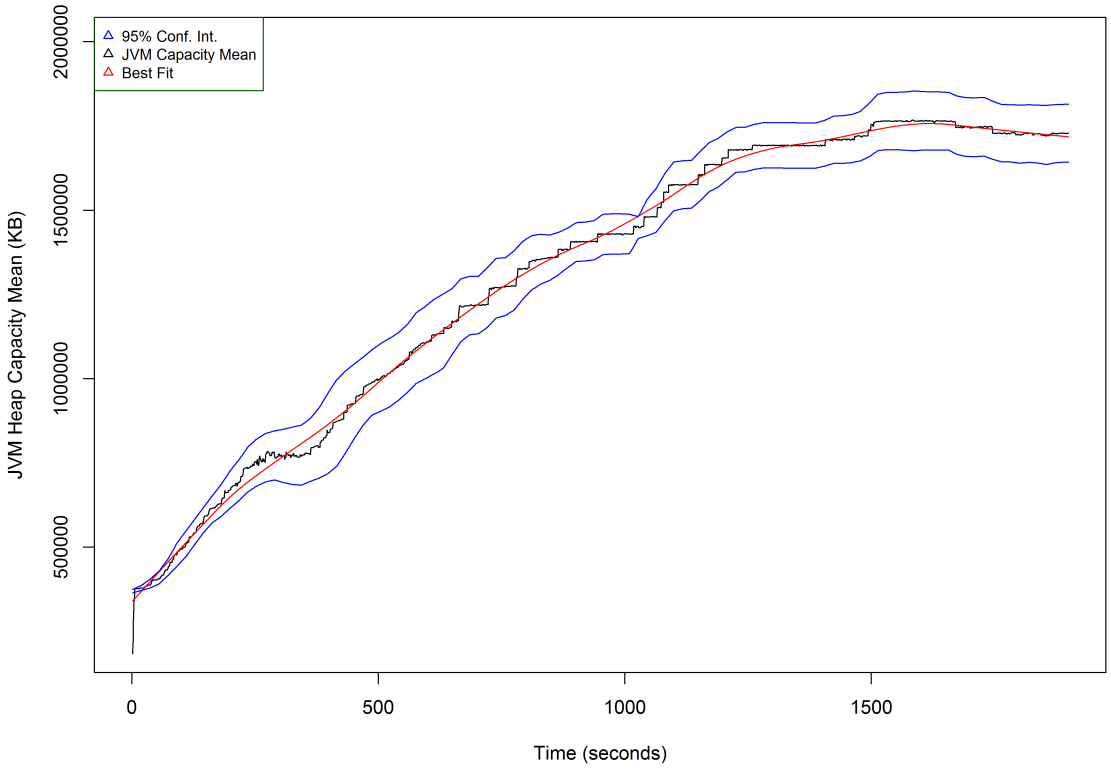

Figure 45. Experiment 1: Switch Attack Mean JVM Heap Capacity

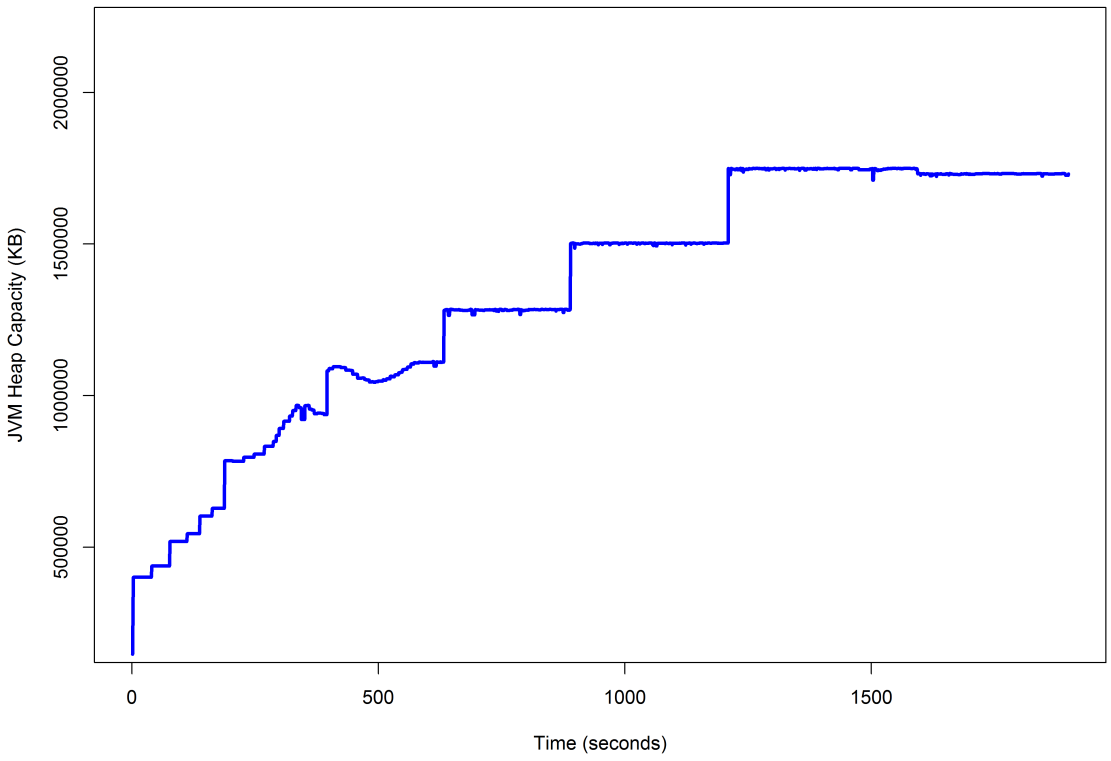

Figure 46. Experiment 1: Switch Attack JVM Heap Capacity Trial 2

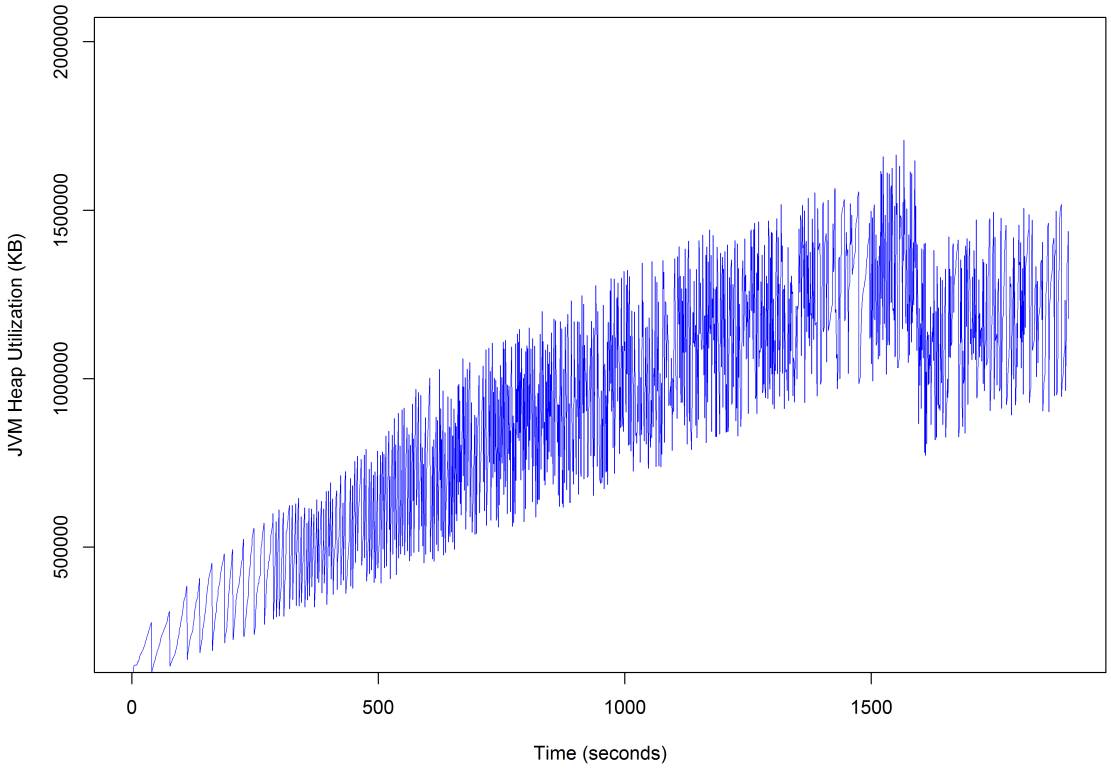

Figure 47. Experiment 1: Switch Attack JVM Heap Utilization Trial 2

Appendix C. Controller Memory Attack Additional Graphs

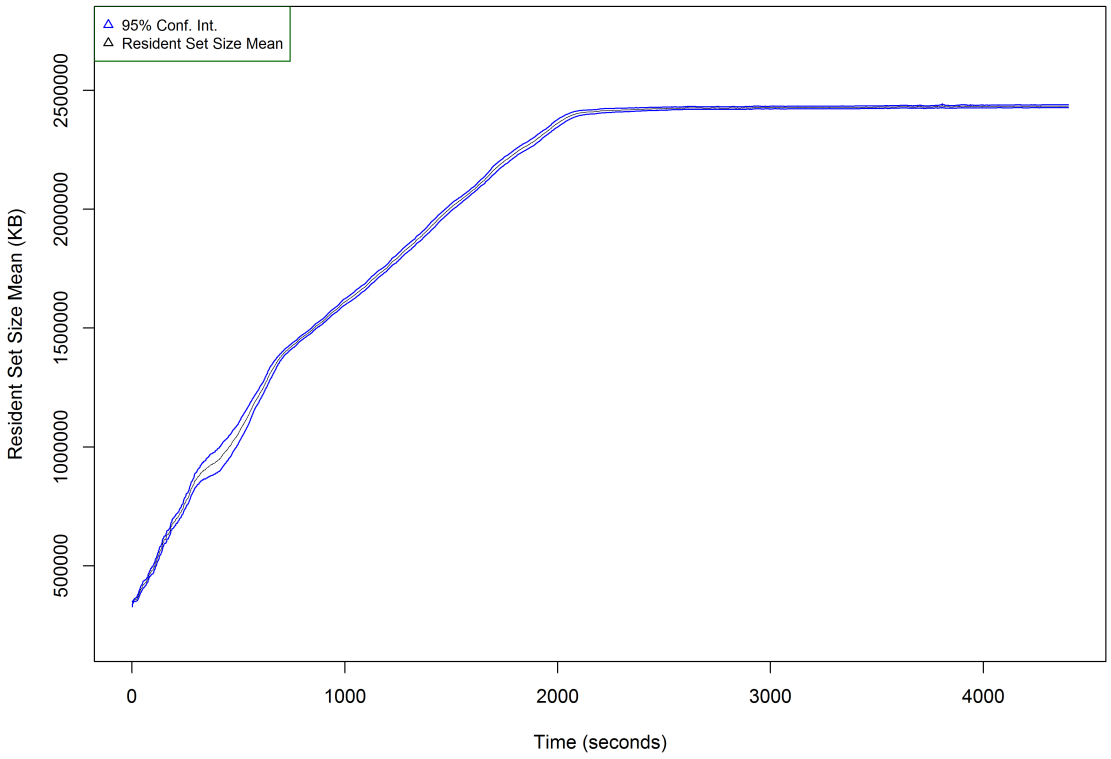

Figure 48. Experiment 2: Controller Memory Attack Mean RSS Memory

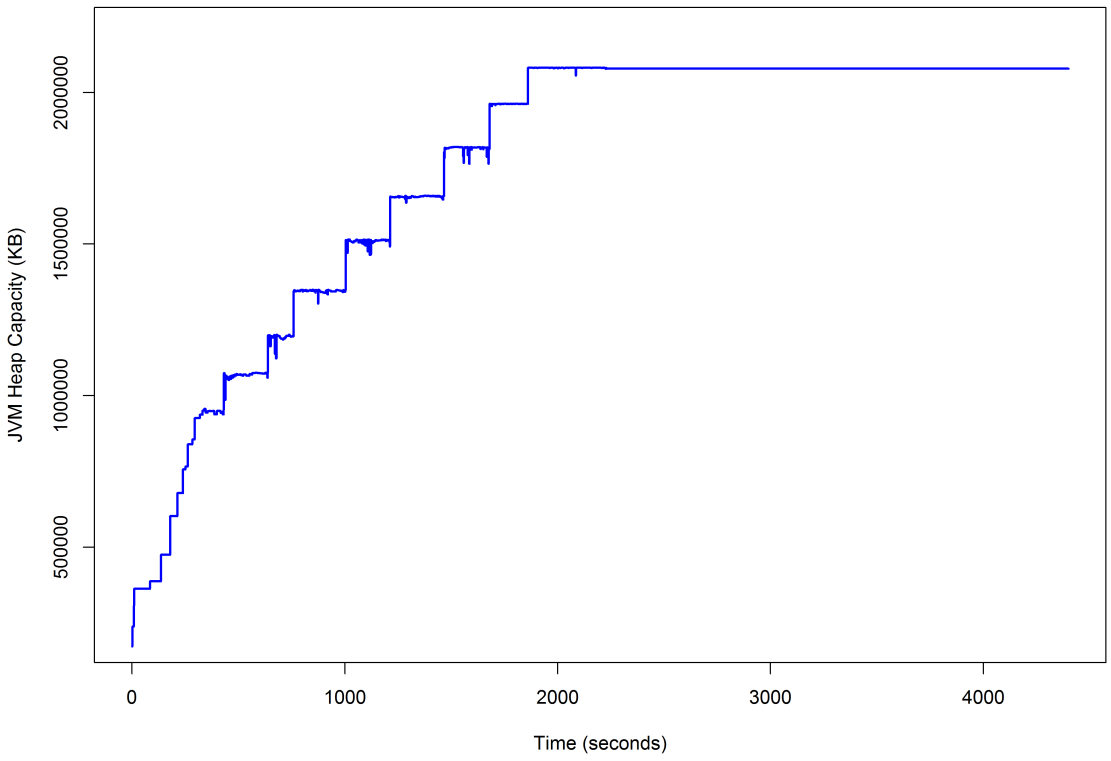

Figure 49. Experiment 2: Controller Memory Attack JVM Heap Capacity Trial 2

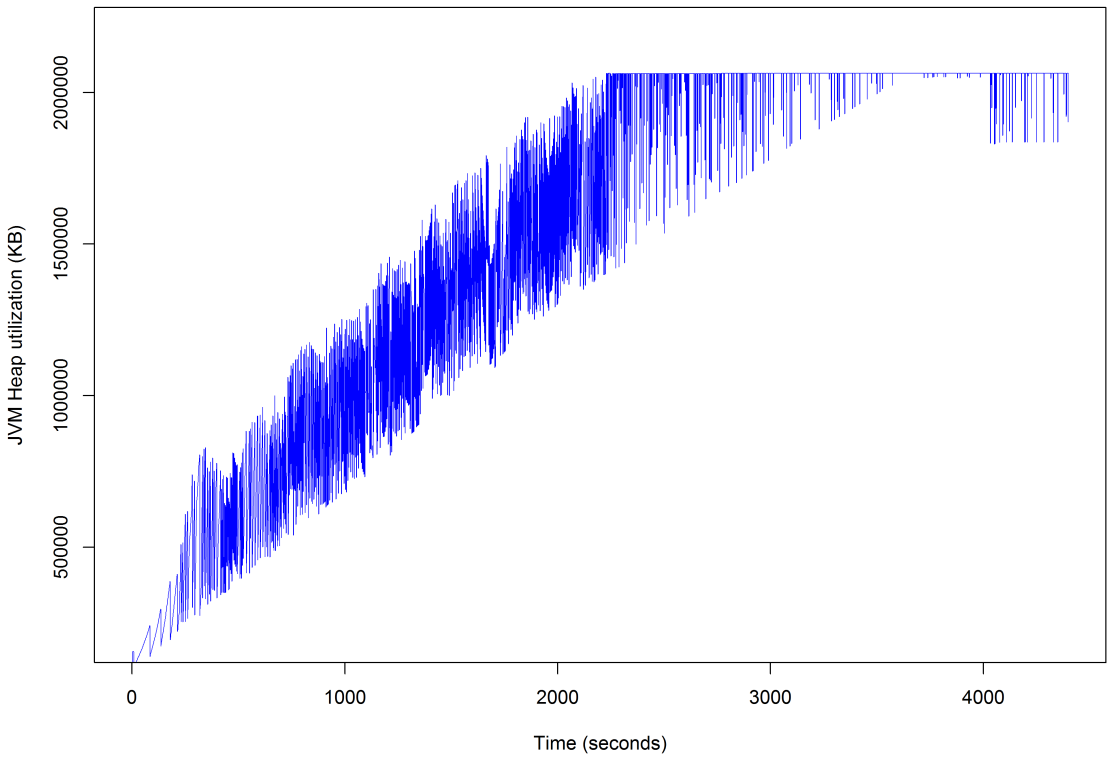

Figure 50. Experiment 2: Controller Memory Attack JVM Heap Utilization Trial 2

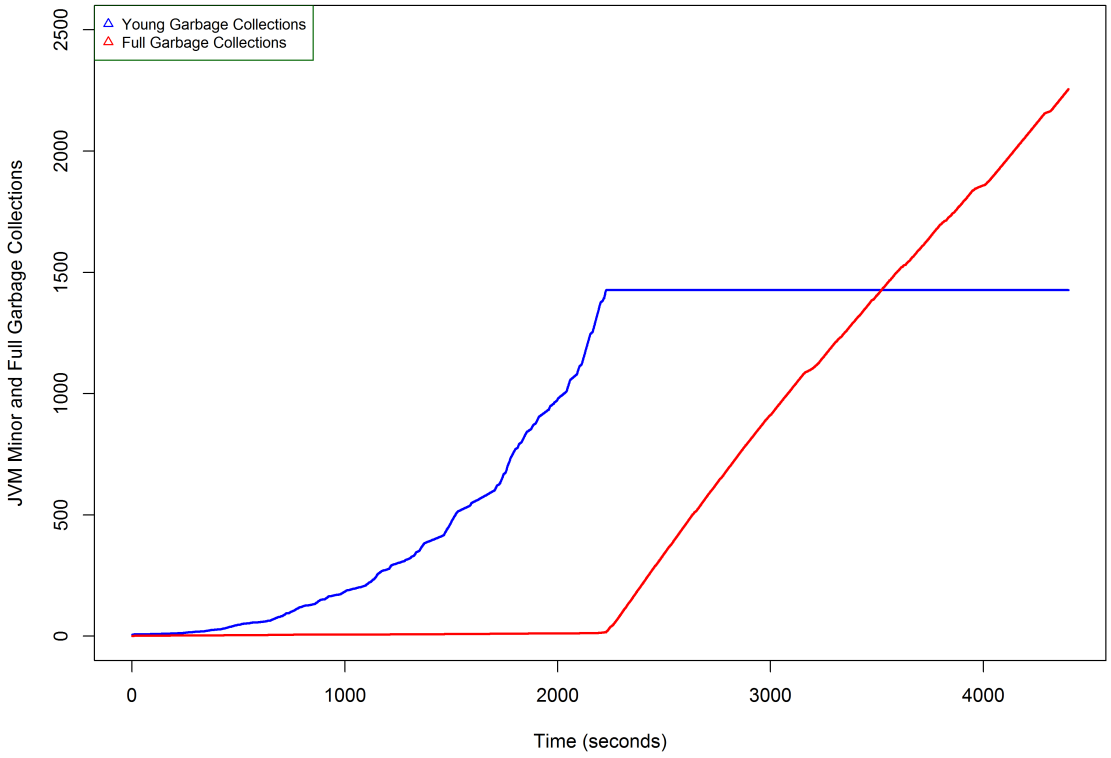

Figure 51. Experiment 2: Controller Memory Attack Young versus Full Garbage Collections Trial 2

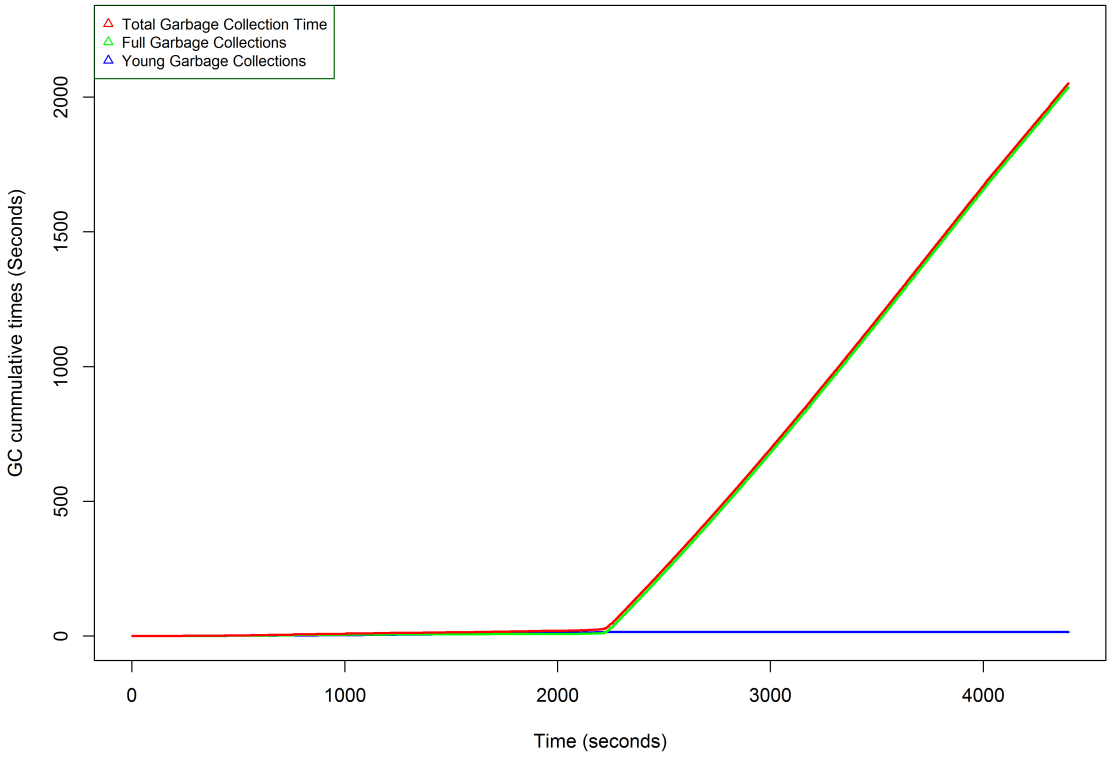

Figure 52. Experiment 2: Controller Memory Attack Garbage Collection Times Trial 2

Bibliography

1. Mandiant. Apt1: Exposing one of china's cyber espionage units. Accessed: https://www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf on 25 May, 2016.

2. Symantec Corporation. Internet security threat report. Accessed: https:// www.symantec.com/ content/ dam/ symantec/ docs/ reports/ istr-21-2016-en.pdf on 25 May, 2016.

3. Open Networking Foundation. Software define networking: The new norm for networks. Accessed: https:// www.opennetworking.org/ images/ stories/ downloads/ sdn-resources/ white-papers/ wp-sdn-newnorm.pdf on 15 Aug, 2016.

4. Project Floodlight. Floodlight documentation. Accessed: https:// floodlight.atlassian.net/ wiki/ display/ floodlightcontroller/ Getting+Started on 20 November, 2016.

5. Open Networking Foundation. Openflow switch specification. Accessed: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.3.pdf on 29 November, 2016.

6. Flowgrammable. Finite state machine models. Accessed: http:// flowgrammable.org/ sdn/ openflow/ state-machine/ on 29 November, 2016.

7. Oracle Learning Library. The jvm and java garbage collection - oracle learning library live (recorded webcast event). Accessed: https://www.youtube.com/watch?v=DoJr5QQYsl8 on 21 November, 2016.

8. J. Pettit J. Luo N. McKeown M. Casado, M. Freedman and S. Shenker. Ethane: Taking control of the enterprise. *ACM SIGCOMM Computer Communications Review*, pages 1–12, 2007.

9. Big-Switch Networks Incorporated. Floodlight. Accessed: http:// www.projectfloodlight.org/ floodlight/ on 12 January, 2017.

10. OpenDaylight Project. Gettingstarted: Development environment setup. Accessed: https://wiki.opendaylight.org/view/GettingStarted:Development_Environment_Setup on 12 January, 2017.

11. S. Donaldson, S. Siegel, C. Williams, and A. Aslam. *Enterprise Cybersecurity: How to Build a Successful Cyberdefense Program Against Advanced Threats*. Apress, 2015.

12. P. Dowd and J. McHenry. Network security: It's time to take it seriously. Accessed: https://www.computer.org/csdl/mags/co/1998/09/r9024.pdf on 25 May, 2016.

13. B. Daya. Network security: History, importance, and future. Accessed: http://web.mit.edu/bdaya/www/Network%20Security.pdf on 11 Novemeber, 2016.

14. Microsoft Corporation. Common types of network attacks. Accessed: https:// technet.microsoft.com/ en-us/ library/ cc959354.aspx on 25 May, 2016.

15. D. Piscitello and ICANN. Threats, vulnerabilities and exploits - oh my! Accessed: https://www.icann.org/news/blog/threats-vulnerabilities-and-exploits-oh-my on 26 May, 2016.

16. E. Cole and S. Ring. *Insider Threat: Protecting the Enterprise from Sabotage, Spying, and Prevent Employees and Contractors from Stealing Corporate Data*. Syngress, 2006.

17. J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach*. Pearson Higher Ed, forth edition edition, 2013.

18. A. Tanenbaum and D. Wetherall. *Computer Networks: Pearson New International Edition: University of Hertfordshire*. Pearson Higher Ed, forth edition edition, 2013.

19. M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. A retrospective on evolving sdn. *Procedings of the First Workshop on Hot Topics in Software Defined Networks*, pages 85–90, 2012.

20. Open Networking Foudation. Open networking foudation overview. Accessed: https://www.opennetworking.org/about/onf-overview on 20 September, 2016.

21. M. Todd United States Air Force. Dynamic network security control using software defined networking. Master's thesis, U.S. Air Force Institute of Technology, 2016. Accessed: https://www.afit.edu/docs/AFIT-ENG-MS-16-M-049.pdf on 20 November, 2016.

22. Oracle Corporation. The java virtual machine. Accessed: https:// docs.oracle.com/ javase/ specs/ jvms/ se7/ html/ jvms-1.html on 21 November, 2016.

23. J. Masamitsu and Oracle Corporation. Garbage collection - let the vm do it. Accessed: https://blogs.oracle.com/jonthecollector/entry/our_collectors.html on 22 November, 2016.

24. Oracle Corporation. Java platform, standard edition hotspot virtual machine garbage collection tuning guide. Accessed: https:// docs.oracle.com/ javase/ 8/ docs/ technotes/ guides/ vm/ gctuning/ sizing.html on 30 November, 2016.

25. K. Benton, L.J. Camp, and C. Small. Openflow vulnerability assessment. *ACM HotSDN*, pages 151–152, 2013.

26. I. Ahmad, S. Namal, M. Ylianttila, and A. Gurtov. Security in software defined networks: A survey. *IEEE Communication Surveys and Tutorials*, 2015.

27. R. Kandoi and M. Antikainen. Denial-of-service attacks in openflow sdn. *IFIP/IEEE IM 2015 Workshop: 1st International Workshop on Security for Emerging Distributed Network Technologies (DISSECT)*, pages 1322–1326, 2015.

28. G. Pickett. Abusing software defined networks. Accessed: https:// defcon.org/ images/ defcon-22/ dc-22-presentations/ Pickett/ DEFCON-22-Gregory-Pickett-Abusing-Software-Defined-Networks-UPDATED.pdf on 1 June, 2016.

29. Oracle Corporation. Garbage collector ergonomics. Accessed: http:// docs.oracle.com/ javase/ 8/ docs/ technotes/ guides/ vm/ gc-ergonomics.html on 21 November, 2016.

30. Oracle Corporation. Understand the outofmemoryerror exception. Accessed: https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html on 23 November, 2016.

31. Open Networking Foundation. Openflow switch specification v1.4. Accessed: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf on 9 December, 2016.

32. Pica8 Open Networking. Pica8 p-3290 datasheet. Accessed: http:// www.pica8.com/ documents/ pica8-datasheet-48x1gbe-p3290-p3295.pdf on 1 December, 2016.

33. M. Maechler and E. Zurich. Scatter plot smoothing. Accessed: https:// stat.ethz.ch/ R-manual/ R-patched/ library/ stats/ html/ lowess.html on 11 December, 2016.

34. W. Cleveland. Robust locally weighted regression and smoothing scatterplots. *Journal of the American Statistical Association*, 74(368):829–836, 1979.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
23–03–2017	Master's Thesis	Aug 2015 — Mar 2017

4. TITLE AND SUBTITLE

Analysis of Denial-of-Service Attack Vectors in Software-Defined Networks

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Portante, Anthony, A., 2d Lt, USAF

5d. PROJECT NUMBER

17G139

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-MS-17-M-060

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Software Defined Networking is a new emerging technology that is quickly gaining popularity amongst the largest corporations. However, this new networking paradigm has a centralized point of failure at the controller. With this choke point, it is imperative that it be designed with security at the forefront. This research aims to shed light on one of the possible ways that having a centralized point of failure in the network can provide malicious attackers an avenue to disrupt an entire enterprise network. Two experiments are performed. The first experiment confirms a discovered vulnerability in a hardware switch. The second, to see if generating fake malicious software switches on the network is enough to inflate the Java Virtual Machine Heap to capacity and cause the controller to crash.

15. SUBJECT TERMS

Software-Defined Networking, SDN, OpenFlow, Java Virtaul Machine, JVM, Denial-of-Service, DoS, Network Attack, Security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Barry E. Mullins (ENG)
U	U	U	U	106	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x7979 Barry.Mullins@afit.edu